NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

USEFULNESS OF COMPILE-TIME RESTRUCTURING OF LGDF PROGRAMS IN THROUGHPUT-CRITICAL APPLICATIONS

by

David M. Cross

September, 1993

Thesis Co-Advisor, ECE Dept — Shridhar B. Shukla
Thesis Co-Advisor, CS Dept — Amr Zaky

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Department of Electrical and Computer Engineering, Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable): EC	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5121	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
USEFULNESS OF COMPILE-TIME RESTRUCTURING OF LARGE GRAIN DATA FLOW PROGRAMS IN THROUGHPUT-CRITICAL APPLICATIONS (U)

12. PERSONAL AUTHOR(S): Cross, David M.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 9/91 TO 9/93	September 1993	56

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Revolving Cylinder (RC), Start-After-Finish (SAF), Start-After-Start (SAS), Large Grain Data Flow (LGDF) Systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
In this thesis, Large Grain Data Flow (LGDF) representation of parallelism is applied to throughput-critical applications that process periodically arriving data. The applications are represented by directed acyclic graphs in which a vertex represents an indivisible node program execution and an arc represents data flow from its source node to sink node. The machine and graph parameters are assumed to be such that the time to transfer one unit of data is comparable to the time to execute one operation at a processor. The machine model consists of a set of processors connected to a set of memory modules by a cross-bar interconnection network. Execution of LGDF graphs on such machines either requires a run-time mechanism to dispatch executable nodes on available processors or a compile-time static scheduling of nodes to processors. The former approach, although flexible and robust, suffers from contention-related overhead and the latter, although capable of eliminating contention, is rigid and computationally intensive.

It is shown by simulation that throughput can be improved when compile-time graph restructuring is coupled with simple first-come-first-serve dispatching. The restructuring is based on selectively adding control dependencies between graph nodes. This technique, called the revolving cylinder analysis, is shown to be an effective framework for achieving communication / computation overlap and reducing memory contention.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Shridhar B. Shukla	22b. TELEPHONE (Include Area Code): (408) 656-2764	22c. OFFICE SYMBOL: EC/Sh

Approved for public release; distribution is unlimited.

Usefulness of Compile-Time Restructuring of Large Grain Data Flow Programs

in Throughput-Critical Applications

by

David M. Cross
Captain, United States Army
B.S.E.E., Rensselaer Polytechnic Institute, 1986
M.S.B.A., Boston University, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Author: David M. Cross
David M. Cross

Approved By: B Shukla
Shridhar B. Shukla,
Thesis Co-Advisor, ECE Dept

Amr Zaky
Amr Zaky
Thesis Co-Advisor, CS Dept

Michael A. Morgan
Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering

Abstract

In this thesis, Large Grain Data Flow (LGDF) representation of parallelism is applied to throughput-critical applications that process periodically arriving data. The applications are represented by directed acyclic graphs in which a vertex represents an indivisible node program execution and an arc represents data flow from its source node to sink node. The machine and graph parameters are assumed to be such that the time to transfer one unit of data is comparable to the time to execute one operation at a processor. The machine model consists of a set of processors connected to a set of memory modules by a cross-bar interconnection network. Execution of LGDF graphs on such machines either requires a run-time mechanism to dispatch executable nodes on available processors or a compile-time static scheduling of nodes to processors. The former approach, although flexible and robust, suffers from contention-related overhead and the latter, although capable of eliminating contention, is rigid and computationally intensive.

It is shown by simulation that throughput can be improved when compile-time graph restructuring is coupled with simple first-come-first-serve dispatching. The restructuring is based on selectively adding control dependencies between graph nodes. This technique, called the revolving cylinder analysis, is shown to be an effective framework for achieving communication / computation overlap and reducing memory contention.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

The modern military depends on real-time digital signal processing applications (such as radar and sonar). These applications generate huge amounts of data continuously. Most of the data is of a time-critical nature which must be processed quickly and accurately. Advances in computer technology have made it much easier to analyze this data. However, the signal processing applications are constantly improving also, generating even more data more quickly.

Large Grain Data Flow (LGDF) graphs can be used to represent these applications. Data flow graphs not only describe the dependencies between different parts of the computation required in an application, but also provide built-in scheduling and synchronization. For example, on a hypothetical system with no communication cost and an unlimited number of processors, nodes can synchronize by sending data and a node can be scheduled as soon as all the required data is present at its input. Due to the generality of this representation, it can be used to specify parallelism at the instruction level [Ref. 1] as well as at the task level [Ref. 2]. The theoretical foundation for the consistency of such representations has been well studied [Ref. 3, Ref. 4].

In practical implementations of this paradigm, the machine must provide mechanisms to manage the data that flows through the graph and to capture the intrinsic scheduling and synchronization. These mechanisms, typically operating at run-time, result in overhead that leads to suboptimal performance. The amount of overhead depends critically on the granularity of the parallelism expressed by the graph and on whether the computations have conditionals and recursion. A direct implementation in hardware of the data flow paradigm for general applications results in unmanageable overheat [Ref. 1, Ref. 5].

Any data flow implementation must perform buffering and fetching of data, allocation of graph nodes to processors, their ordering on each, and the exact times at which they are scheduled. If all the related decisions are done at run-time, the efficiency of the implementation suffers. The overhead can be reduced effectively by using the node and arc attributes of the data flow graph at compile-time to simplify the run-time management. Based on which decisions are made at compile-time and which ones are made at run-time, data flow implementations can be classified over a spectrum that ranges from fully static to fully dynamic [Ref. 6]. While dynamic implementations have more overhead, they are more flexible and are easier to implement. They also degrade gracefully in the event of individual processor malfunction. On the other hand,

static implementations are more efficient and lead to predictable performance which is crucial to real-time systems. However, they are difficult to realize, are inflexible, and do not degrade gracefully. Their effectiveness is determined by how accurately the computational requirements of the application are known. This is typically a difficult problem and its solution of using the worst-case estimate can result in large inefficiencies.

Therefore, real-time systems must strike a careful balance between the compile-time effort and run-time complexity to get the desired and guaranteed performance. For classes of applications, such as signal processing, such balance can be obtained by exploiting two properties of the computations required, the availability of a priori knowledge of the amount of data produced and consumed and negligible use of conditionals and recursion. When the amounts of data produced and consumed by the nodes of a data flow graph are known exactly, the applications are called synchronous data flow applications [Ref. 2]. When the data arrives periodically, they have been classified as pipelined function-parallel computations [Ref. 7]. In real-time signal processing applications, the trade-off between compile-time and run-time has an additional dimension because of the periodic arrival of data. When external data arrives periodically, the intrinsic non-determinism of data flow execution results in unpredictable program behavior. As a result, processed data arrives unpredictably leading to the possibility of intolerable delays and insufficient buffer space, especially under high loads.

A. THESIS SCOPE AND CONTRIBUTION

The focus of this work is on compile-time mechanisms for controlling data flow execution. A technique, called revolving cylinder (RC) analysis originally introduced in [Ref. 8], in which, instead of generating information, such as schedules, to control allocation or ordering on processors at run-time, a new data flow graph is obtained at compile-time which gives a better throughput and behaves more predictably than the old graph under the same run-time mechanism. The key idea in restructuring based on RC analysis is that inserting dependencies in the graph can produce a graph with better performance. This idea can be traced back to algorithms for overlapping complex operations on pipelined processors [Ref. 9]. This restructuring selectively changes the conditions when a node will enter the list of executable nodes; however, choosing the processor to schedule it on is left to the run-time dispatcher. This enables the actual scheduling to remain dynamic keeping the run-time overhead low.

This thesis defines a model for a Large Grain Data Flow system, which is loosely based on the Department of the Navy AN/UYS-2 Digital Signal Processing System (also known as the Enhanced Modular

Signal Processor, EMSP) [Ref. 10]. Baseline results will be generated to show that it is possible to improve the system throughput over that offered by first-come-first-serve (FCFS) scheduling by compile-time restructuring of the LGDF programs following the RC technique. The utility of several computer programs designed to analyze this LGDF model and FCFS and RC scheduling will be verified with the generation of the results.

B. THESIS ORGANIZATION

Chapter II describes fully the LGDF system model. Included are descriptions of the hardware and software, along with the joint hardware/software view. Chapter III is a description of the FCFS and RC scheduling techniques. Chapter IV is an analysis of the data generated for the LGDF model using all the scheduling techniques presented. Chapter V summarizes the results and presents possible topics for future study.

C. ADDITIONAL RESEARCH

Additional results and further analysis of the concepts in this thesis are included in [Ref. 11]. The computer programs used to generate the results in this thesis are described in detail with complete examples and program listings in [Ref. 12].

II. THE LARGE GRAIN DATA FLOW MODEL

A Large Grain Data Flow (referred to as LGDF) computer system can be defined in terms of the two major categories which are used to define most computer systems, hardware and software.

A. SOFTWARE MODEL

The software model of a data flow system is usually visualized as a graph. There are two primary elements to this data flow graph, nodes and queues. There are five secondary elements to the graph, system input nodes, system output nodes, system input queues, system output queues, and synchronization arcs. These secondary elements are necessary for the computer program which models this system. Figure 2.1 is a simple data flow graph example showing the graph symbols. Note that there are no special symbols for system input and output queues, they are determined by their attachment to the system input and output nodes.

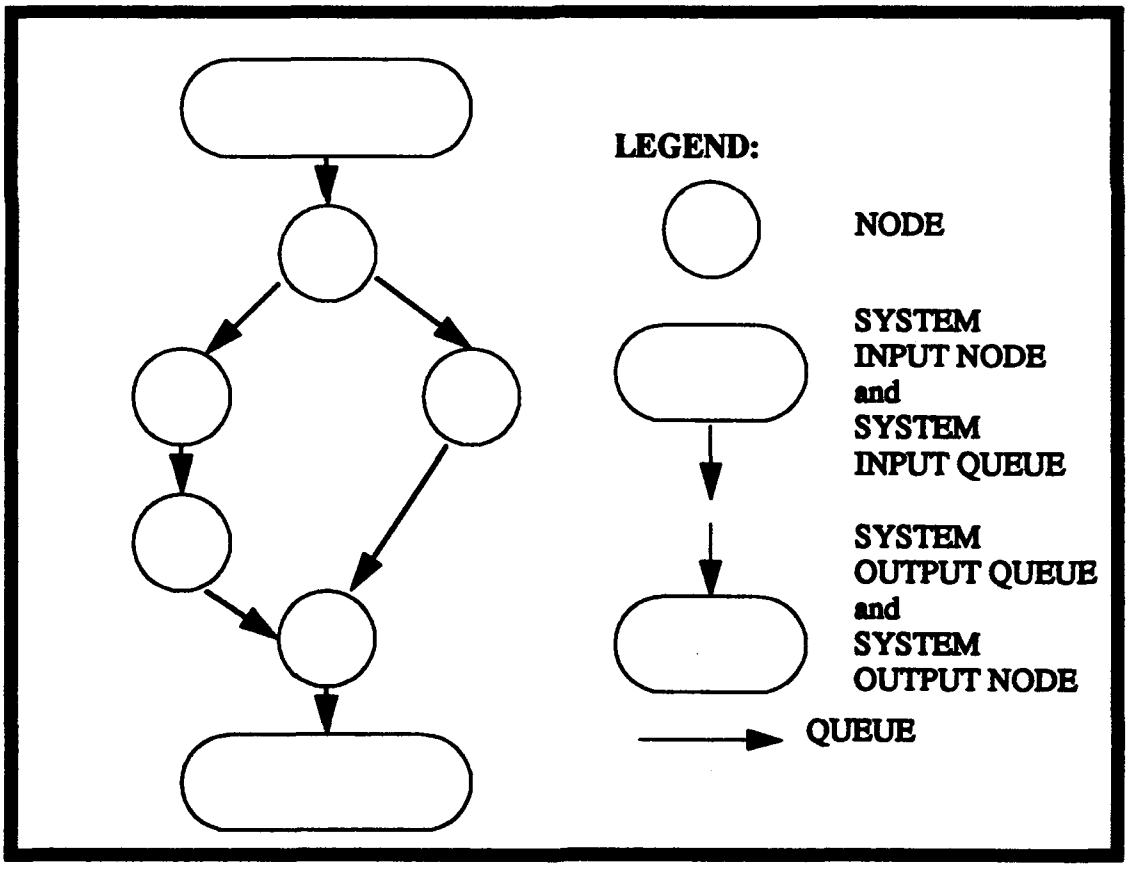

Figure 2.1. Data Flow Graph Example

1. Terms

There are several important terms which will be defined here.

a. Cycle

The term 'cycle' is used to describe an arbitrary time unit. It could represent any unit of time, but is usually interpreted as a microsecond.

b. Word

The term 'word' is used to describe an arbitrary data element. In the model, it could represent any unit of data size, but is usually interpreted as a byte.

c. Processing

'Processing' refers to all activities performed by a node on a processor. This includes actual node execution, the transfer of information between the processor and memory (both instruction and data), and any latency.

d. Execution

'Execution' refers only to the actual execution of the node on a processor to accomplish a given task. It does not include any memory operations or latency involved with those operations.

e. Input and Output

The terms 'input' and 'output' are used in many varied contexts. To eliminate the confusion, any reference to the beginning and end points into the graph are referred to as 'system' inputs and outputs.

2. Nodes

Nodes represent software modules which perform a specific function. This module could be a program or a subroutine or a function. What is inside the node is not important to model the LGDF system. The model is only concerned with the length of time it will take the node to complete its given operation and the amount of data input into the node and output from the node.

In this model, a node is characterized by several parameters.

a. Execution Time

The execution time (in cycles) is the time required by the node to complete its function once the data and the node instructions have been loaded onto a specific processor.

b. Setup Time

The setup time (in cycles) represents a constant latency before a node is able to access any memory modules after being assigned to a processor.

c. Breakdown Time

The breakdown time (in cycles) represents a constant latency for the node that has completed memory operations before the processor is made available in the free processor pool.

d. Instruction Size

The instruction size is listed in words. The instruction size is used to determine how long it will take to load the code segment represented by the node to a processor for execution. This time is dependent on the data transfer rate of the hardware.

e. Processor Type

The processor type is used to specify nodes which must use a specific type of processor.

3. Queues

Queues are used to represent the flow of data. Each queue connects a pair of nodes, and the amount of data transferred between the nodes is identified. Data is transferred from the node at the tail of the queue (named the source node) to the node at the head of the queue (named the sink node).

In this model, a queue is characterized by several parameters.

a. Threshold Amount

The threshold amount is the amount of data (in words) required to be on the queue for the sink node to begin execution.

b. Produce Amount

The produce amount is the amount of data (in words) added to the queue upon completion of one execution instance of the source node.

c. Consume Amount

The consume amount is the amount of data (in words) removed from the queue upon the start of one execution instance of the sink node.

d. Write Amount

The write amount is the amount of data (in words) written from the source node to memory upon completion of one execution instance.

e. Read Amount

The read amount is the amount of data (in words) read by the sink node from memory prior to the beginning of one execution instance.

f. Capacity

The capacity is the total amount of data (in words) which can be stored on the queue. If the capacity of the queue would be exceeded, a source node cannot produce any more data until the sink node consumes data to open space on the queue.

g. Initial Length

The initial length is the amount of data (in words) is placed on the queue at system start-up.

h. Relationship among the Parameters

There are several important distinctions to be made between the parameters. It would appear that the produce and write amounts are equivalent and the consume and read amounts are equivalent. For most data queues, the produce and write amounts would be the same quantity as would consume and read amounts. However, the functions performed are distinctly different. The read and write amounts represent actual data transfers required between a processor and memory. These transfers require a large amount of time to complete. The produce and consume amounts represent a control function within the scheduler. No data is actually transferred but the queue length recorded by the scheduler is adjusted. The difference would become more obvious when synchronization arcs are discussed.

There is one major requirement to be met by the parameters. This requirement is that the capacity of the queue must be greater than or equal to the threshold. If this is not the case, then there could never be enough data on the queue to cause the sink node to trigger.

For most data queues, the threshold and consume amounts will be the same. This means that the sink node requires a set amount of data to trigger. When this threshold is reached, the sink node will consume that much data in execution.

In many cases, the produce amount will also be the same as the threshold and consume amounts. This represents a linear program. The source node produces the exact amount of data which is

required and used by the sink node. However, this is not always the case. If the produce amount is less than the threshold, then the source node must execute multiple times before triggering the sink node. If the produce amount is greater than the consume amount, the sink node must execute multiple times upon completion of the source node.

Figure 2.2 is a graphical representation of the queue parameters.

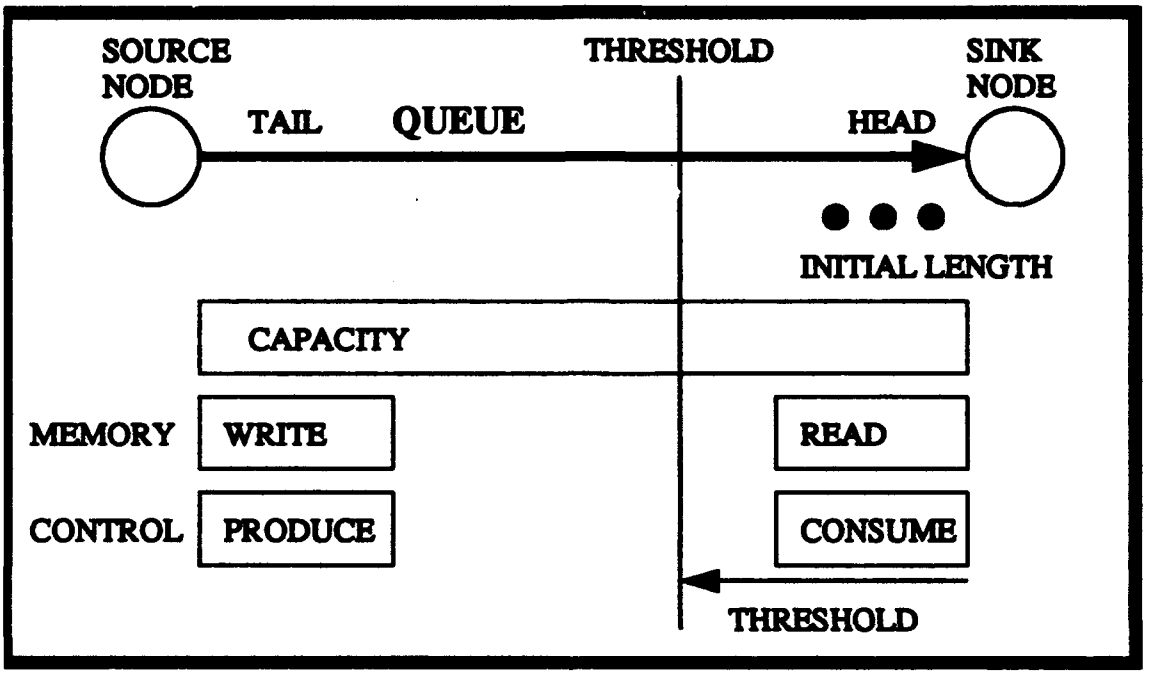

Figure 2.2. Graphical Description of Queue Parameters

4. System Input Nodes and System Input Queues

System input nodes are necessary to simulate multiple execution instances of the graph. Upon initiation of a graph instance, this node is activated. System input nodes have the same parameters as nodes as defined above. However, system input nodes will operate on a special input/output processor. The system input node is the sink node of an external queue. This external queue does not really exist, but functions as a queue with infinite capacity and a threshold and consume amounts of one data unit. When the graph instance is initiated, one data unit is produced onto this queue. The output queues from the system input nodes are designated system input queues. They function exactly as the data queues described above. However the data written to them comes from an I/O processor.

5. System Output Nodes and System Output Queues

System output nodes are necessary to simulate multiple execution instances of the graph. Once all the queues into this node have exceeded threshold, this node is activated. System output nodes have the same parameters as nodes as defined above. However, system output nodes will operate on a special input/output processor. The system output node is the source node of an inherent queue. This inherent queue does note really exist, but functions as queue with infinite capacity. As this system output node is executed, it can be assumed that all input queues to this node transfer the data equal to the consume amount to the outside as data output. The input queues to the system output nodes are designated system output queues. They function exactly as the data queues described above. However the data read by them is read by an I/O processor.

6. Synchronization Arcs

Synchronization arcs are a special subclass of the queues described above. However, they function slightly differently. They represent control signals which will be described later. Due to the control nature of these arcs, the produce and consume amounts are generally one, representing a counter. However, the read and write amounts will always be zero. This is because the synchronization arcs reside only in the scheduler memory, and no data is actually ever transferred to a processor. The threshold and initial length amounts are highly variable depending upon the RC analysis and used to trigger nodes in a specific order.

B. HARDWARE MODEL

The Large Grain Data Flow system is a multiprocessor system. The major component of the system is the arithmetic processor. Additional components modeled include the input/output processor, global memory modules, the scheduler, and the data transfer network. Figure 2.3 is a diagram of the LGDF hardware model.

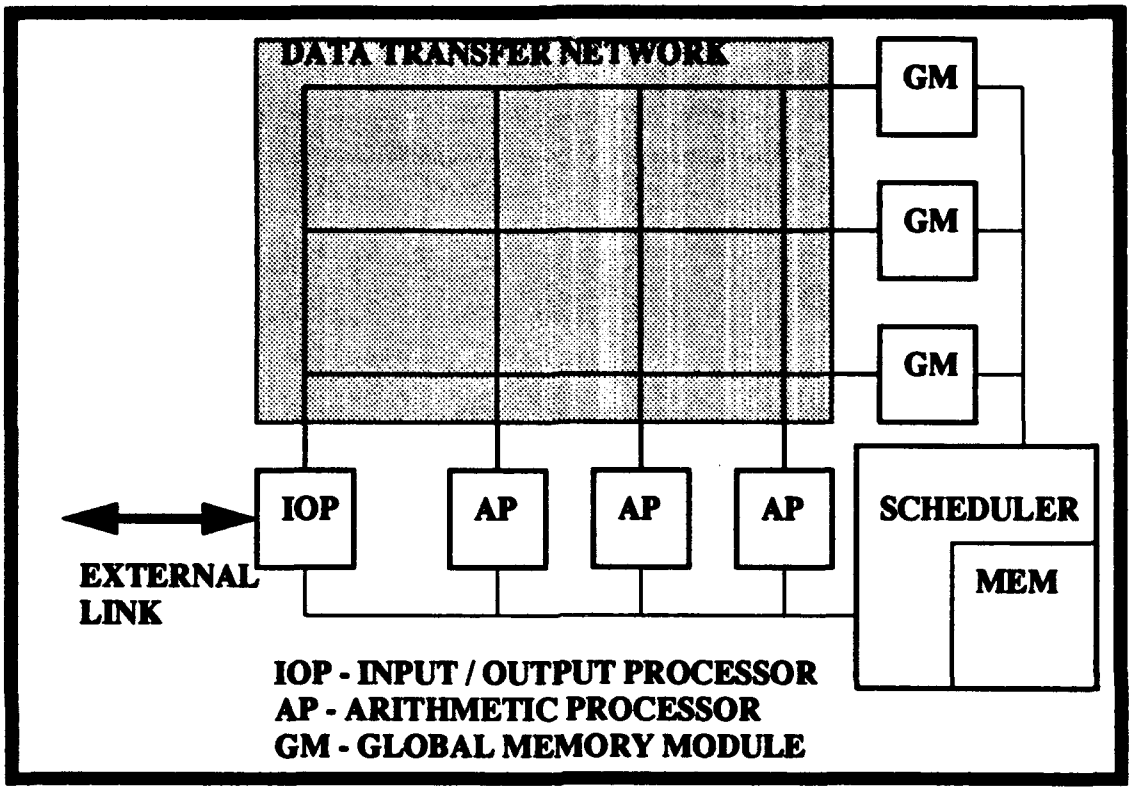

Figure 2.3. Large Grain Data Flow Hardware Model

1. Arithmetic Processor

The arithmetic processors in this model consist of two units, the execution unit and the control unit. The nodes complete their tasks on the execution unit. All communications and setup and breakdown latency are handled by the control unit. Two nodes can be processing on a given processor during a given time. One node can be doing a task on the execution unit. The other node can be on the control unit, either preparing to execute when the execution unit is available, or removing itself from the processor and writing results when finished execution. The arithmetic processors are assumed to be sophisticated, able to control many instructions and manipulate large amounts of data on the chip. This means that no data is transferred during the processing of a node, only before and after execution.

2. Input/Output Processor

The input/output processor acts no differently from the arithmetic processors described above. However, it only handles the system input and output nodes and system input and output queues. Data is transferred into and out of the system through this processor. The input/output processor does not factor into utilization statistics.

3. Scheduler

The scheduler is the unit which tracks the entire system state. It also acts as a memory controller, maintaining a table of all the instruction and data locations, tracking the queue levels to decide when to trigger nodes. It is assumed the scheduler has sufficient internal memory to track all of the system information. A scheduler latency time, expressed in cycles, can be assigned to abstractly represent the time it takes the scheduler to change the state of its local memory when the amounts on a queue are adjusted.

4. Global Memory Module

The system main memory is modeled as a series of modules. These modules are considered global since they can be accessed by any processor. A processor must obtain control over the appropriate memory module to access a queue for either a read or write operation, or to load a node instruction. This information is supplied to the processor by the scheduler. Multiple module accesses can progress simultaneously; however, at any time, only a single processor can access a given memory module. The size of the memory is assumed large enough to meet any requirements. Nodes and queues can be assigned to specific memory modules by the user or arbitrarily by the scheduler.

5. Data Transfer Network

The data transfer network is an abstraction in this model. It is assumed that all transactions between all current processor and memory module pairings can proceed. No transaction will be delayed because the network is busy. Thus, the data transfer network acts as a full crossbar switching network. There is a constant data transfer time to transfer one word of data between the processor and memory. This is known as the word communications time expressed in cycles per word.

C. OVERALL SYSTEM MODEL

Sections A and B above describe the software and hardware specifics. To define the overall system, the interaction of the software and the hardware must be considered. This can best be displayed by considering the software and hardware perspectives of what is actually happening. The node and the processor are the elements chosen for these perspectives.

1. Node Perspective

The node is the primary software element. An LGDF system is designed so that a node, when all the data is available, can be assigned to any available processor of the type that the node requires. A ready list is maintained of all the nodes which are ready to execute.

The scheduling unit knows the structure of the entire data flow graph and can track the status of all nodes and queues. These events are between the node and the scheduler: Check if Data is Available, Check if Data Space is Available, Check if Processor is Available. The rest of the events are between the node and the assigned processor.

a. Check if Data is Available

The scheduling unit checks each queue which has the node as a sink. If all of the queues which enter the node are above threshold, then the node is 'input' ready.

b. Check if Data Space is Available

The scheduling unit checks each queue which has the node as a source. If all the queues have enough space below capacity to receive the data produced when the node completes, then the node is 'output' ready. The node is now assigned to the node ready list.

c. Check if Processor is Available

The node ready list is a First-Come-First-Serve wait list. The scheduler moves along the list from head to tail and checks for each node in the list if the proper type of processor is available. If a processor of the proper type for the node is available, the node is assigned to that processor.

d. Setup

The node begins preparation for execution as specified in the node setup latency parameter (in cycles). The node is utilizing the processor control unit.

e. Load Instruction

The node loads the code segment from memory to the control unit. This is specified by the node instruction length parameter (in words) and the word communication time (cycles per word) along with any delay in accessing the memory unit where the instructions reside.

f. Read Data / Consume Data

The node proceeds to read the data from the appropriate queues, up to the specified read amount parameter for the queues. The scheduler simultaneously consumes data from the queues up to the specified consume amount parameter. The time spent for each queue is specified by the read parameter (in words) and the word communication time (cycles per word), along with the scheduler latency time (in cycles). Additionally, delays could result if the memory unit where the data is stored is currently being used by another processor. This event is not complete until the information for all input queues has been read and/or consumed.

g. Check for Execution Unit Availability

Once the data queues are read, the node is ready for execution. However, the execution unit might be in use by another node. Thus, the node may be blocked, waiting on the execution unit. Once the execution unit becomes available, the node will switch from the control unit to the execution unit.

h. Execute

The node performs execution as specified by the node execution time parameter (in cycles).

i. Check for Control Unit Availability

Once the node has completed execution, it is ready to output the results and remove itself from the processor. However, the control unit might be in use by another node. Thus, the node may be blocked, waiting on the control unit. Once the control unit becomes available, the node will switch from the execution unit to the control unit.

j. Write Data / Produce Data

The node proceeds to write the data to the appropriate queues, up to the specified write amount parameter for the queues. The scheduler simultaneously produces data to the queues up to the specified produce amount parameter. The time spent for each queue is specified by the write parameter (in words) and the word communication time (cycles per word), along with the scheduler latency time (in cycles).

Additionally, delays could result if the memory unit where the data is stored is currently being used by another processor. This event is not completed until the information for all output queues has been written and/or produced.

k. Breakdown

The node removes itself from the processor as specified by the node breakdown latency parameter (in cycles). Upon completion of breakdown, the node is disassociated from the processor. This completes one entire iteration for a node.

l. Summary

Table 2.1 provides a summary of the above listed events and the proper calculation of their processing times. The term 'delay' refers to stalls caused by memory conflicts, the inability to access a queue or instruction in memory due to that memory module being used by another node.

Table 2.1: PARAMETER DEFINITIONS

Code	Definition / Time
ExecutionTime	Node Execution Time Parameter (in cycles)
SetupTime	Node Setup Latency Time Parameter (in cycles)
BreakdownTime	Node Breakdown Latency Time Parameter (in cycles)
InstLen	Node Instruction Length Parameter (in words)
WriteAmt	Queue Write Amount Parameter (in words)
ReadAmt	Queue Read Amount Parameter (in words)
CommTime	Word Communications Time (in cycles per word)
LatencyTime	Scheduler Latency Time (in cycles)
LoadTime	CommTime * InstLen + delays
ReadTime	[(LatencyTime + CommTime * ReadAmt) + delays] for all queues with the node as a sink
WriteTime	[(LatencyTime + CommTime * WriteAmt) + delays] for all queues with the node as a source

m. Event Reductions

Most all of the events result in a time mark for the next event. Therefore, several of the events can be combined to simplify the model. Many of these events, although different, contribute to an overall time which lends itself to easier analysis of the results. The resulting event reductions are defined as phases for easy differentiation with the previously described events.

(1) Input Phase - This event represents the total time a node spends on the control unit for a given iteration, from the time it is assigned to the time the execution unit becomes available. It includes these events: Setup, Load Instruction, Read Data / Consume Data, and Check for Execution Unit Availability.

(2) Execution Phase - This event represents the total time a node spends on the execution unit for a given iteration, from the time the execution unit becomes available to the time the control unit becomes available. It includes these events: Execute, and Check for Control Unit Availability.

(3) Output Phase - This event represents the total time a node spends on the control unit for a given iteration, from the time the control unit becomes available to the time breakdown is completed. It includes these events: Write Data / Produce Data, and Breakdown.

Table 2.2 is a summary of the time calculations for these phases. The term blockage refers to stalls caused by the inability of a node to switch to the other processing element (control unit to execution unit or execution unit to control unit) until the node on the other processing element completes its operation. It is to be noted that the contention for memory modules during the input and output phases is implicit in 'ReadTime' and 'WriteTime' respectively.

Table 2.2: PHASE TIME DEFINITIONS

Code	Definition / Time
InputTime	SetupTime + Load Time + ReadTime + blockage
ExecuteTime	ExecutionTime + blockage
OutputTime	WriteTime + BreakdownTime + blockage

a. Representation Comparison

Figure 2.4 is a graphical representation of these times as associated with a processor. Two diagrams are given. The first diagram is the detailed model. The second diagram is the reduced model. As far as node scheduling techniques are concerned, the reduced model will be used.

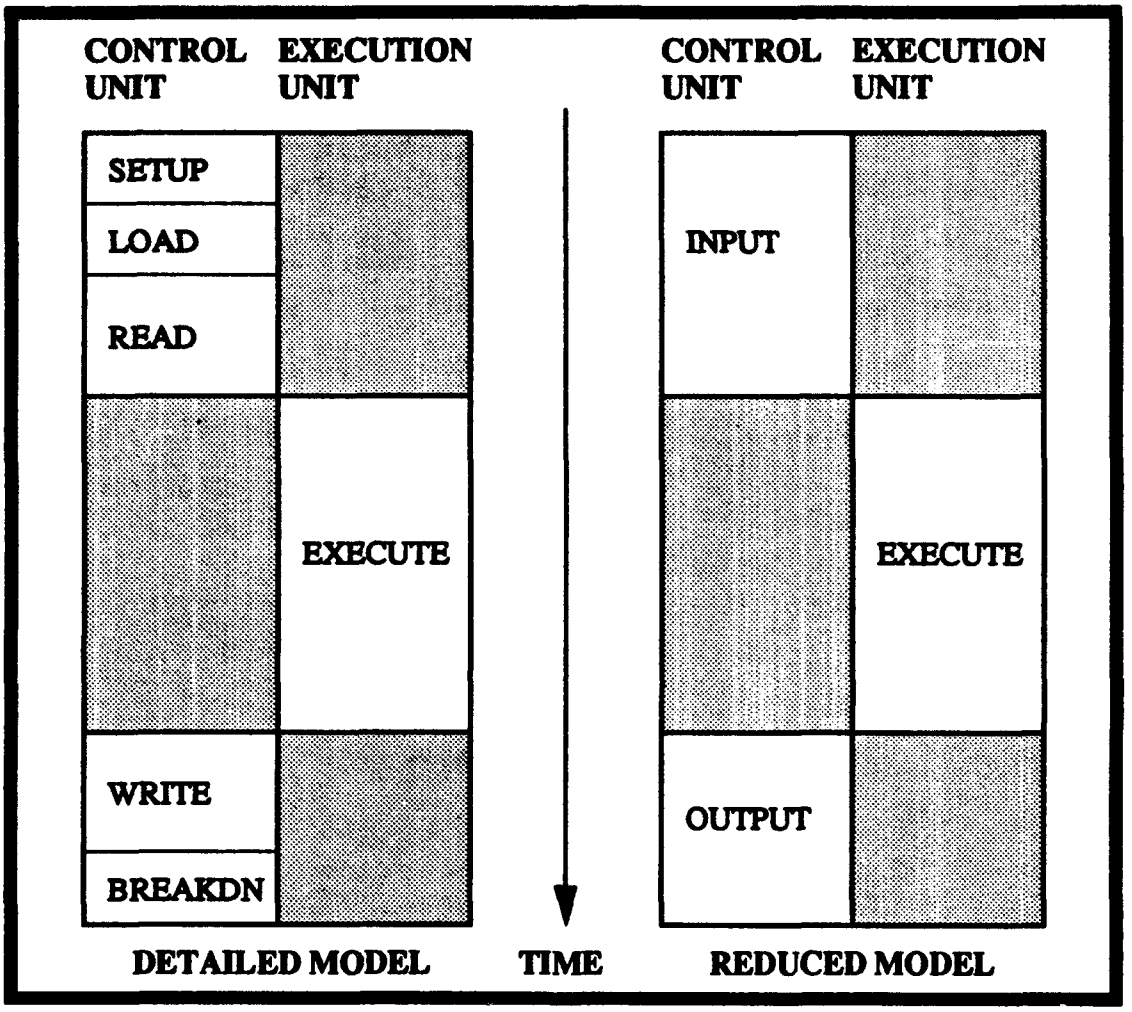

Figure 2.4. Time on Processor Representation

2. Processor Perspective

The processor can be best described as a finite state machine. Two finite state diagrams are given. These state diagrams represent the same system, but from different points of view. Figure 2.5 is the internal view state diagram. This is the state of the processor and nodes as it appears on the processor. Figure 2.6 is the external view state diagram. This is the state of the processor as it appears to the outside world.

Table 2.3 lists the codes used to define the states. Note that one control unit code and one execution unit code are required to define a complete state.

Table 2.3: STATE DIAGRAM CODES

State Code	State Description
ExeFree	Execution Unit is Free
ExeBusy	Execution Unit is Busy (node is in Execution Phase)
ConFree	Control Unit is Free (Processor Available for Node Assignment)
ConBusy	Control Unit is Busy (a node is performing either Input or Output)
ConInput	Control Unit is Busy with a node performing Input
ConOutput	Control Unit is Busy with a node performing Output

Several of the transitions require further explanation. Recall that two different nodes can be operating on a processor at any given time. One node is performing execution on the execution unit, and the other node is performing either input or output on the control unit.

(1) In the case where one node is executing and another is performing input, then neither node can go to the next state until both actions are completed, as the nodes must swap the units they are currently occupying, with the node which completed execution moving to the control unit to perform output and the node which completed input moving to the execution unit to perform execution. This transition is defined as 'Execution and Input Completed'.

(2) In the case where one node is executing and another is doing output, there are two possible occurrences. If the node performing output completes first, then it simply is removed from the processor. However, if the node executing completes first, it stalls while waiting for the other node to complete output. When this second node completes output, it will disassociate itself from the processor and the node which completed execution will obtain use of the control unit. This transition is defined as 'Execution Completes then Output Completes'.

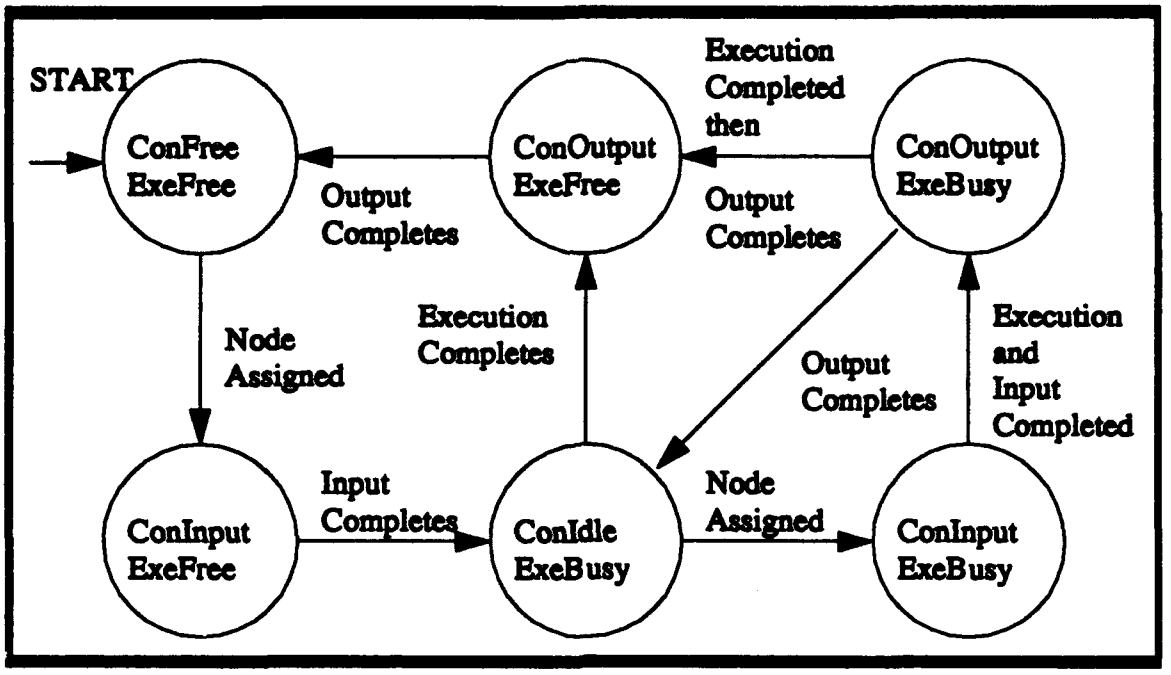

Figure 2.5. Processor Internal View State Diagram

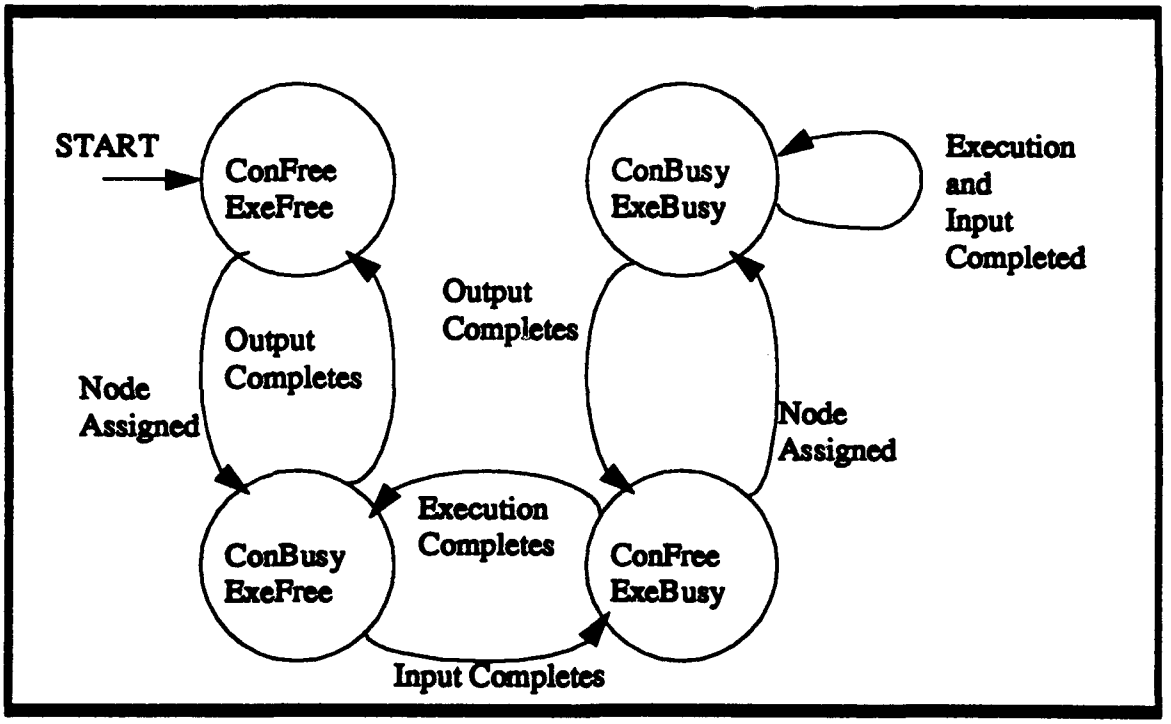

Figure 2.6. Processor External View State Diagram

III. SCHEDULING TECHNIQUES

A key factor in the Large Grain Data Flow (LGDF) system is the scheduling of the nodes in the data flow graph to the processors. This chapter will discuss important scheduling issues inherent to the LGDF model, scheduling techniques, and possible improvements.

A. TERMS

Several important concepts are used in the analysis of the scheduling techniques.

1. Throughput

Throughput is the total number of instances completed in a given time interval. Throughput is uniform if the time interval between the completion of consecutive graph instances is constant.

2. Response Time

The response time is the time it takes to complete one iteration of a graph. This is the actual time from the beginning of graph processing to the end of graph processing for a given graph iteration. The response time is uniform if each graph instance completes in a constant time.

B. COMMUNICATION / COMPUTATION OVERLAP

An important aspect of this LGDF model is the dual unit processors. Each processor has a control unit and an execution unit. Different nodes can be operating simultaneously on different units of the same processor. All communications and node control functions take place on the control unit. It is desirable to have these control and communication functions done concurrently with the execution of another node. This is known as communication / computation overlap. Ideally, the communications and control functions would completely overlap with the execution.

To fully appreciate the techniques, the concept of communication / computation overlap must be introduced. This can best be shown graphically. Previously, Figure 2.4 displayed one node upon a processor. However, in this LGDF model, two nodes will normally be on a processor simultaneously. There are many possible situations which can occur.

Many of these situations are described graphically in detail. Note that these figures display the state of the processor in the middle of activities. The node designated 'node 0' has been executing for some time. 'node 1' has just been assigned to the processor.

In the following descriptions, the term 'communication' represent all communications and control functions and latency times. The term 'computation' represents the actual processor execution. These two terms are selected as they are prevalent in current literature.

1. Perfect Communication / Computation Overlap

Figure 3.1 displays the perfect overlap condition. This condition is rather unrealistic as it is highly unlikely that the communications would perfectly match the computation. However this is the theoretical case.

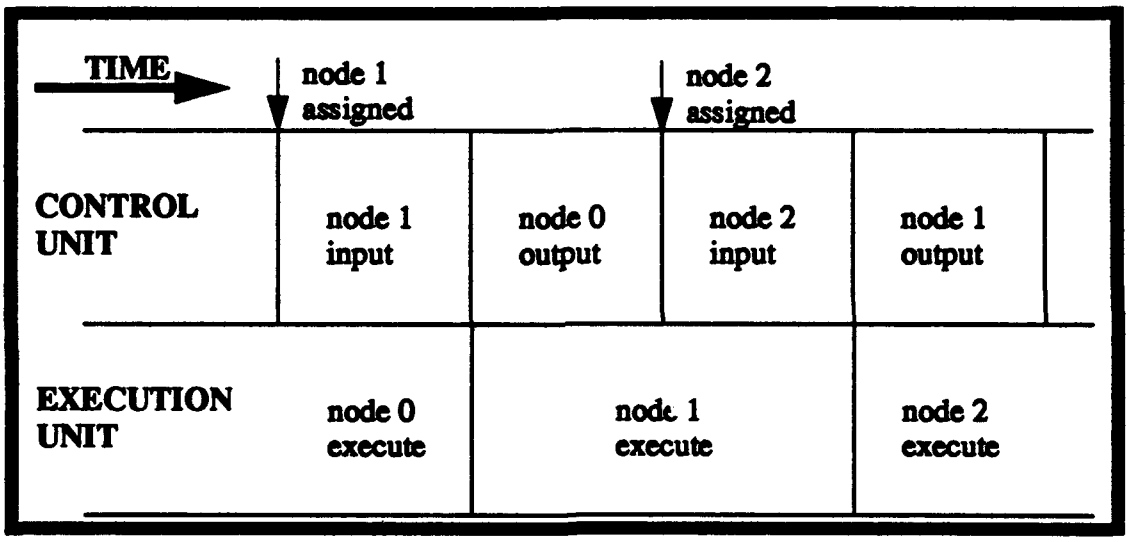

Figure 3.1. Perfect Communication / Computation Overlap

2. Good Communication / Computation Overlap

Figure 3.2 displays good overlap conditions (assuming that perfect overlap will not occur). In this case, communication is completely overlapped with computation. This situation will tend to occur when the memory access speed is fast compared to processor speed, or the instructions represented by the nodes require large amounts of processing compared to the amount of data transfer.

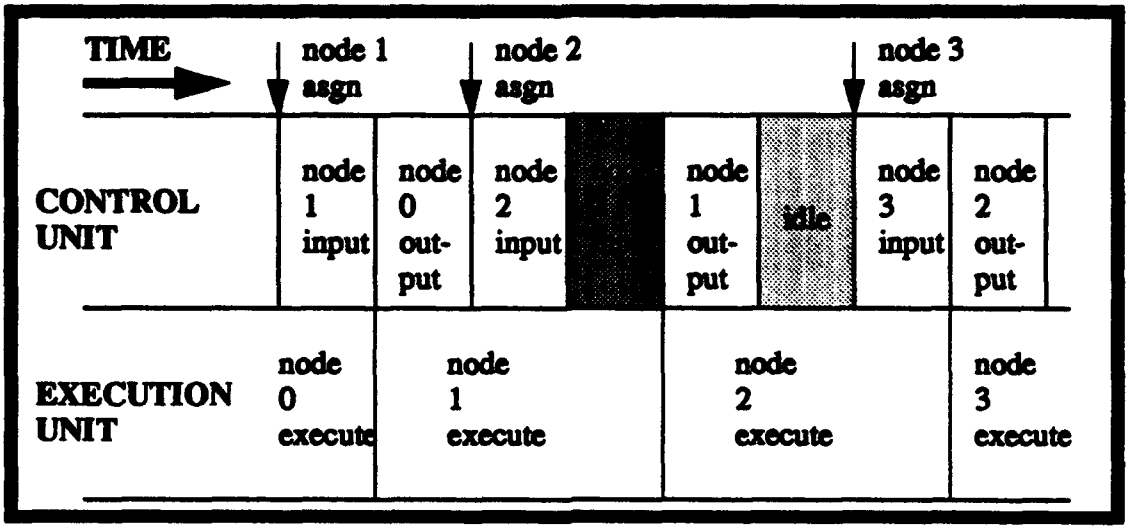

Figure 3.2. Good Communication / Computation Overlap

Several conditions are displayed in Figure 3.2. The heavily shaded portion represents a blocked control unit. Node 2 has completed its input, but it cannot begin execution because node 1 has not completed execution. The lightly shaded portion represents an idle control unit. In this case, no node is ready to begin processing. Neither of these conditions is bad since the execution unit is operating at its full capability.

3. Poor Communication / Computation Overlap

Figure 3.3 displays poor overlap conditions. In this case, communication is not completely overlapped with computation. This situation will tend to occur when the memory access speed is slow compared to processor speed, or the instructions represented by the nodes require small amounts of processing compared to the amount of data transfer.

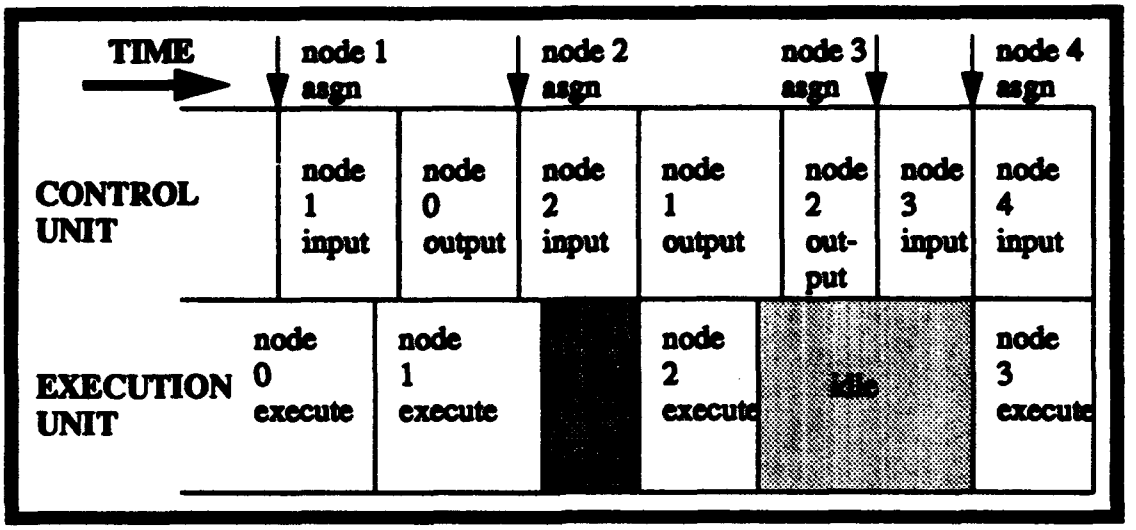

Figure 3.3. Poor Communication / Computation Overlap

Several conditions are displayed in Figure 3.3. The heavily shaded portion represents a blocked execution unit. Node 1 has completed execution, but it cannot commence output until node 2 completes input. The lightly shaded portion represents an idle execution unit. In this case, the control unit is busy forcing the execution unit to be idle. As output has priority over input in the model, the beginning of execution is further delayed until the next ready node completes input. These conditions represent bad performance because no useful execution is being performed.

4. Realistic Communication / Computation Overlap

In actual processing, it is likely that 'good' overlap will occur at times and 'poor' overlap will occur at other times. The various scheduling techniques to be discussed later in this chapter will attempt to force the system to have more 'good' overlap node to processor assignments than 'poor' overlap node to processor assignments. This is not necessarily an easy undertaking as in general, all nodes have wide ranges of execution times and required volumes of communication.

5. Revised Finite State Machine

Figure 2.5 provided a state diagram to describe the processor. With the possible overlap conditions defined in the above diagrams, an expanded state diagram can be provided to more accurately describe the model, provided in Figure 3.4. Table 3.1 provides the processing unit state codes. Once again, an execution unit code and a control unit code are necessary to define the system state.

Table 3.1: STATE DIAGRAM CODES

State Code	State Description
ExeIdle	Execution Unit is Idle
ExeCalc	Execution Unit is Calculating
ExeBlock	Execution Unit is Blocked with a node waiting for the Control Unit
ConIdle	Control Unit is Idle (Processor Available for Node Assignment)
ConInput	Control Unit is Busy with a node performing Input
ConOutput	Control Unit is Busy with a node performing Output
ConBlock	Control Unit is Blocked with a node waiting for the Execution Unit

In node to processor scheduling, it is important to minimize the execution unit idle states (ExeIdle) and execution unit blocked states (ExeBlock). Ignoring the end points of operation (where there must be some execution unit idle time), the goal is to cycle continuously through the following states (this cycle is highlighted on the state diagram):

--> (ConIdle / ExeCalc) -->

--> (ConInput / ExeCalc) -->

--> (ConBlock / ExeCalc) -->

--> (ConOutput / ExeCalc) -->

--> recycle

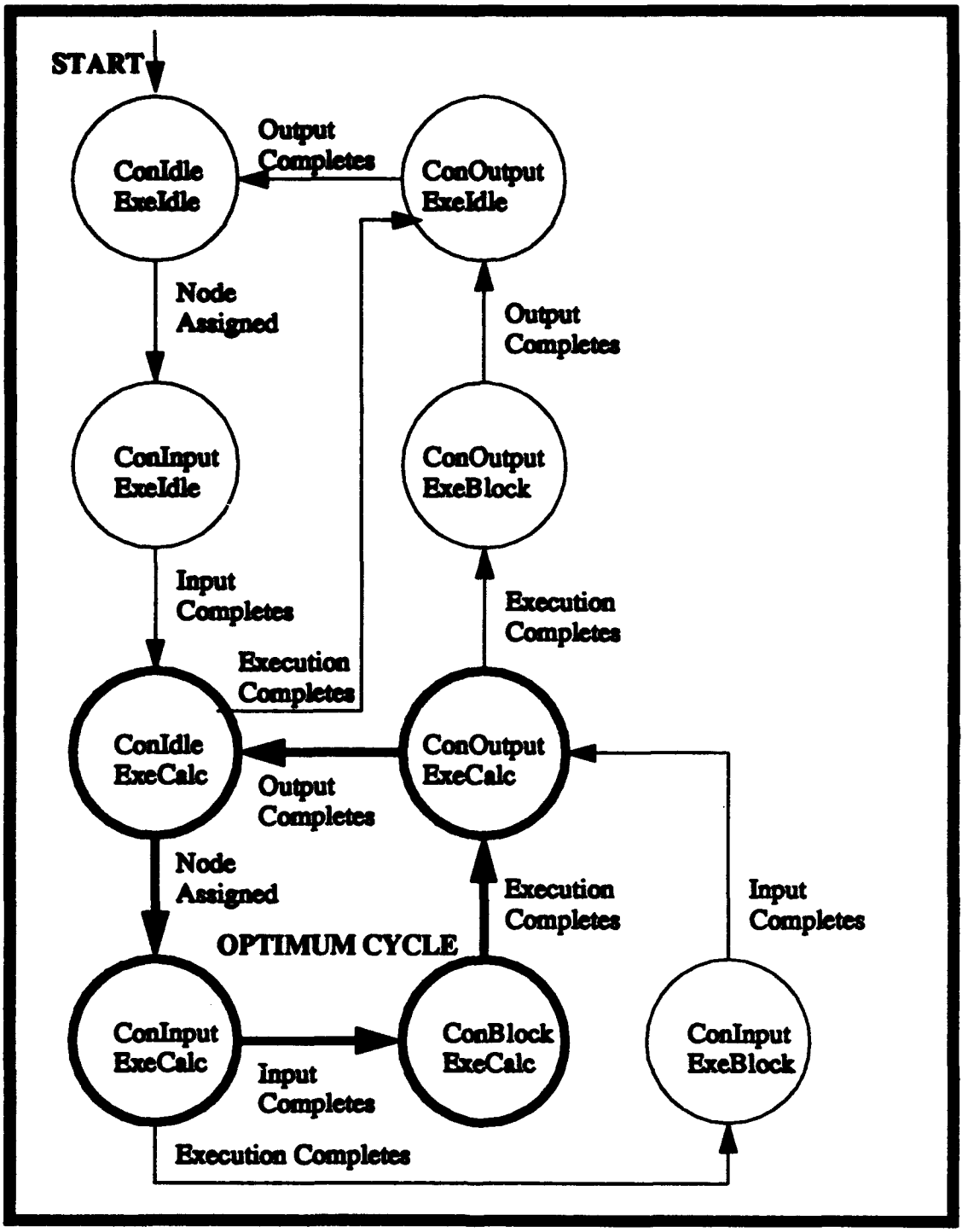

Figure 3.4. Expanded Processor State Diagram

C. CONTENTION

Contention refers to the inability for a communications operation to occur between a processor and a memory module due to the memory module being utilized by another processor. This results in a delay of the node on the processor requesting use of the memory module.

1. Queue Contention

A queue can only be accessed by one node at a time. Therefore, if the source node wants to write data and the sink node wants to read data, one would be delayed until the other completes its current operation.

2. Memory Contention

Memory contention is generally more broad than queue contention, since queue contention represents two nodes trying to access the same set of locations in the memory module. With memory contention, one processor is accessing a node or queue in a specific memory unit. This could be either reading from a queue, writing to a different queue, or loading a node program. While this operation is taking place, no other queue or node program can be accessed by another processor from the same memory module.

D. FIRST-COME-FIRST-SERVE SCHEDULING TECHNIQUE

First-Come-First-Serve (FCFS) scheduling can more properly be stated as a lack of scheduling. Nodes are assigned to processors in the order in which they are made ready. There is no forethought or attempt at optimization.

1. Advantages

a. Simplicity

Since there is no special order to the assignment of nodes, the amount of overhead (software and additional hardware) required for the assignment is negligible.

b. Processor Utilization

Processors will be in use constantly. As long as nodes are in the ready list, they will be assigned to available processors.

c. Minimal Queue Contention

As a function of the FCFS algorithm, the queue contention is minimized. This is due to the fact that a node cannot begin input until all queues into the node are ready. Therefore, the source node will write data to a queue. Then, the queue would be ready to be read by the sink node.

d. Fault Tolerance

With an FCFS implementation of scheduling, the system is fault tolerant. Since nodes will not be assigned to processors until all data is ready, no deadlocks will occur.

2. Disadvantages

a. Communication / Computation Overlap

There is no guarantee of good communication / computation overlap with FCFS, since nodes are placed on the next available processor, regardless if whether the communication times and computation times can be made to overlap.

b. Unpredictable Response Time and Throughput

With the communication / computation overlap that is likely to change from one graph iteration to the next, it is difficult to predict the graph response time and throughput.

c. Memory Contention

Since nodes are assigned to processors when they are ready, there is no way to predict which memory modules would be required at any time.

3. Comments

It can be expected that if communication time is very small compared to computation time for most nodes in the graph, then FCFS can perform well since the effects of the disadvantages will be minimized. Conversely, if the communication time is large compared to computation time, then the disadvantages will be accentuated. We expect the latter to be the case precisely because the graphs are LGDF.

E. REVOLVING CYLINDER SCHEDULING TECHNIQUE

The Revolving Cylinder (RC) scheduling technique is designed specifically for Large Grain Data Flow systems. It is assumed that the application requires the specified data flow graph to be executed continuously.

The premise is that at any given time the nodes of one graph equivalent must be processed. This means that not all of the nodes will be working on the same data set, but one instance of each node is ready to work on a data set. With the RC technique, this one graph equivalent will be mapped to the available processors. This mapping is known as the cylinder. The term revolving cylinder refers to the fact that additional cylinders, exactly the same as the first, can be placed one after another. Essentially, the execution resembles a rotating drum.

There are four variations of the revolving cylinder technique that will be described. The first variation to be presented is Start After Finish (SAF). The second variation, Start After Start (SAS) determines the synchronization arcs in a different manner. In both SAF and SAS, there is no requirement that nodes always be scheduled to the same processor. However, SAF and SAS can be further modified by binding the nodes to specific processors. These variations are termed SAFb and SASb respectively.

1. Index Assignment and Synchronization Arcs

In a given slice, many of the nodes will not be working on the same set of data, therefore, the nodes are assigned an index to reference the data set that node is currently operating on. Once the indices are determined, synchronization arcs are generated. These synchronization arcs are control signals which enforce the cylinder structure.

Figure 3.5 is a simple data flow graph which is scheduled on two processors. Note that for the demonstration of the RC technique, the only node parameter is the execution time. Also note that the input and output nodes do not get mapped to the cylinder. The node identifier is the letter and the node execution time is the number inside the node. In the processor mapping, the index is the number in parenthesis.

Two cylinders are mapped. Indices are assigned to the first cylinder as follows. Ignore the synchronization arcs in determining data dependencies. The first node mapped is 'A'. Therefore, it is given an index of '0'. Nodes 'B' and 'D' depend on the results of 'A'. Node 'B' appears after node 'A' on the same processor. Therefore, it can work on the same data set as 'A', hence an index of '0'. However, node 'D' begins processing at the same time as node 'A'. Since it depends on the results of 'A', node 'D' must be operating on a previous data set, hence an index of '-1'. Node 'C' depends only on node 'B' for data. Although it is scheduled to a different processor, node 'C' does not start until node 'B' completes, therefore, it can still

operate on the same data set as 'B', thus an index of '0'. Node 'E' depends on data from both nodes 'C' and 'D'. It is assigned an index of '-1' for two reasons. First, node 'D' has an index of '-1'. Node 'E' starts after 'D', so it can have the same index, '-1'. Second, node 'C' is processing at the same time as node 'E'. Therefore, node 'E' must be operating on a previous set of data. Therefore, since 'C' has an index of '0', then 'E' must have an index of '-1'. The second cylinder is mapped in the same manner as the first, but with the indices increased by one.

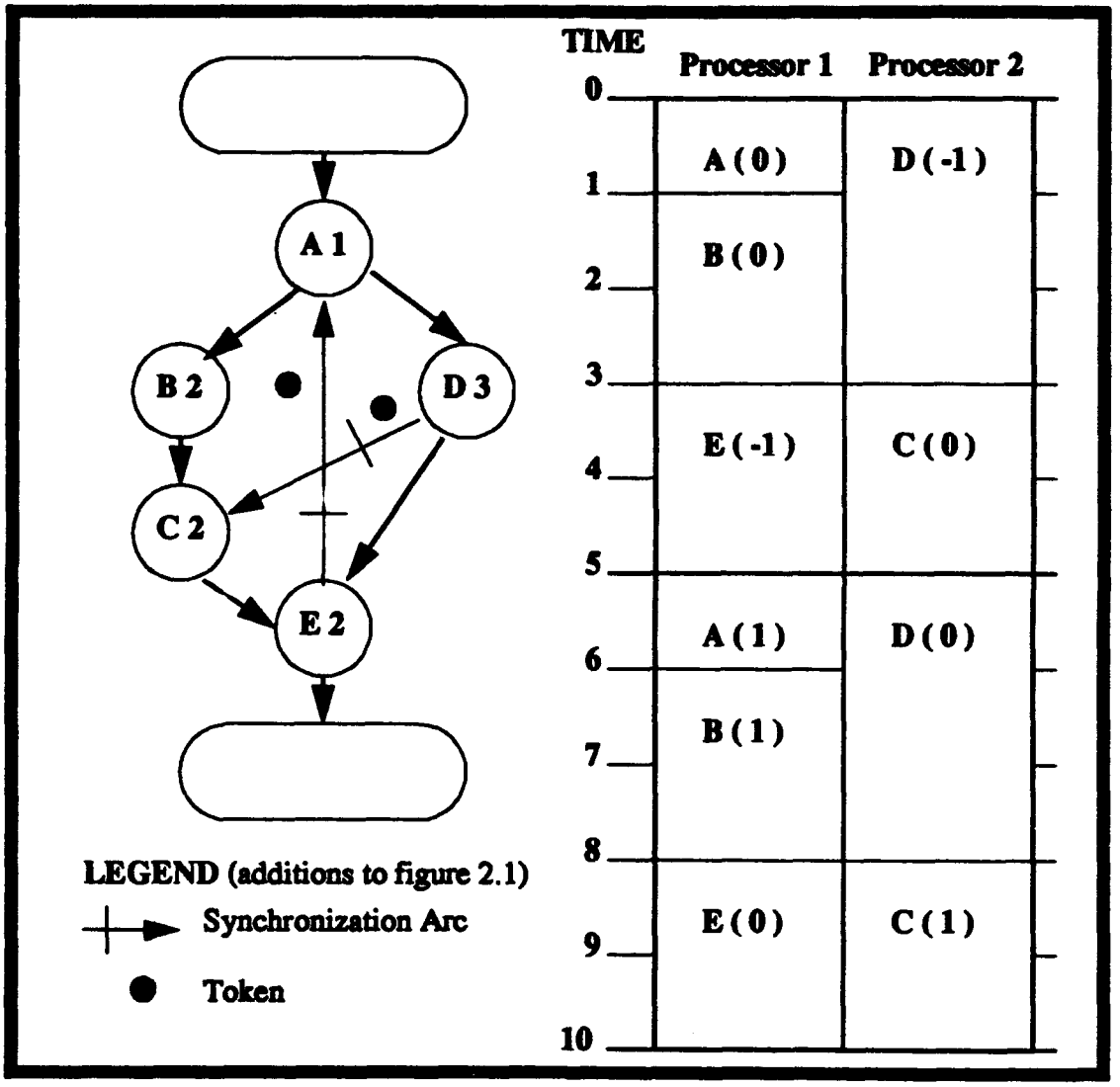

Figure 3.5. Data Flow Graph and Processor Assignment

This is the Start After Finish (SAF) version of the revolving cylinder technique for generating the synchronization arcs. The sink node at the head of the synchronization arc will be allowed to start after the source node at the tail of the arc completes. The synchronization arcs are generated as follows. Nodes 'A',

'B' and 'C' operate in consecutive order on the same instance. Therefore, they maintain data dependence and no synchronization arcs are necessary between them. Likewise, nodes 'D' and 'E' maintain such a data dependence. However, in this mapping, node 'C' executes on the same process as node 'D'. To set up the cylinder, node 'C' must wait for one instance of node 'D' to execute. Therefore, a synchronization arc is generated between 'C' and 'D'. Looking at the whole cylinder, node 'A' cannot start executing until node 'E' of the previous instance completes. Therefore, a synchronization arc exists between 'E' and 'A'.

Tokens on synchronization arcs represent a counter. The tokens listed represent the initial length parameter of the synchronization arc as defined in the previous chapter in the section on queues. It is obvious that these tokens are needed. The synchronization arcs define the need for node 'E' to complete before node 'A' begins. However, no instance of 'E' can ever occur until one instance of node 'A' executes. Therefore, the initial token will allow the process to start. Likewise for the token on the synchronization arc between nodes 'D' and 'C'. After multiple instances of the graph have executed, the cylinder should look as it is with all nodes at the proper index.

Showing two cylinders back to back illustrate some important concepts of the RC algorithm. First, it takes a number of cylinders to complete a graph iteration. This quantity is equal to the range of different indices in the cylinder. The required time is equal to the number of cylinders multiplied by the time to complete one cylinder. In this example, two cylinders are required. Note that the range of indices is two (from 0 to 1). Therefore, the time to complete one graph instance is ten cycles (two cylinders multiplied by five cycles to complete a cylinder). Note that this is longer than the minimum possible time to complete the graph on two processors which is seven cycles (based on longest path) in this example. However, it is guaranteed that it will take ten cycles to complete each and every instance. It is also guaranteed that one instance will complete during each cylinder. In this example, one iteration completes every five cycles. Therefore, the revolving cylinder technique results in uniform throughput and uniform response time.

The above example is rather simplistic and not representative of the Large Grain Data Flow model studied. In the LGDF model, the nodes are not operating in distinct blocks. One node actually begins preparing to execute on a processor before the previous operating node is finished. Therefore, determining the actual indices and arcs is not a simple matter on even a moderately complex data flow graph. However, the start after finish synchronization arc generation and revolving cylinder assignment technique is still valid.

2. Advantages

a. Predictable Performance

Since uniform cylinders are assigned to the processor set, the system will have more predictable throughput and average response time.

b. Maximize Communication / Computation Overlap

The nodes in the cylinder can be placed to achieve maximum overlap of communication time with computation time. If the communication cost of the system is low, there will be little gain to the revolving cylinder technique.

c. Reduce Memory Contention

Once the cylinder is mapped, it can be determined which nodes and queues must be accessed at the same time. Therefore, nodes and queues can be mapped to different memory modules to ensure that they are not active at the same time, reducing memory contention. This could be a difficult task as queues are operated on by different nodes at different times. However, any reduction of memory contention will help. This is impossible with FCFS as it is never known which operations will proceed at any given time.

3. Disadvantages

a. Increased Overhead

Overhead is significantly increased with the requirement to generate and track the synchronization arcs. Also, it is important to generate proper tokens on the synchronization arcs to assure that deadlocks will not occur due to dependencies which cannot be met.

b. No Overlap Between Cylinder Slices

In this LGDF model, all nodes have some input and some output time. However, with the start after finish technique, the first node in the next cylinder cannot begin processing until the last node in the current cylinder completes. Thus, there is no possible communication / computation overlap between cylinder instances.

c. Unbalanced Loads

A related issue to the non-overlap between cylinder instances is the issue of unbalanced loads. An ideal cylinder would have the processors completely load balanced. That is, all processors would

complete processing at the same instant. However, this is usually not the case. The next cylinder cannot begin processing until the last node of the current cylinder completes processing. Therefore, if the last node on one processor completes long after the nodes on the other processors, the additional processors would remain idle for extended periods and the throughput reduced.

d. Queue Contention

Queue contention can be minimized through proper mapping. However, it is now a factor to be taken into consideration.

4. Alternate Revolving Cylinder Scheduling

An alternate version of the revolving cylinder technique, Start After Start (SAS), generates the synchronization arcs based on when a source node node begins, rather than after it ends. This eliminates the lack of communication / computation overlap between consecutive cylinder mappings.

IV. RESULTS AND ANALYSIS

This chapter is an analysis of the initial results for the use of the Revolving Cylinder algorithm. The programs used to generate the results are described fully in [Ref. 12]. Figure 4.1 is a diagram of the relationship of the programs used to generate the results.

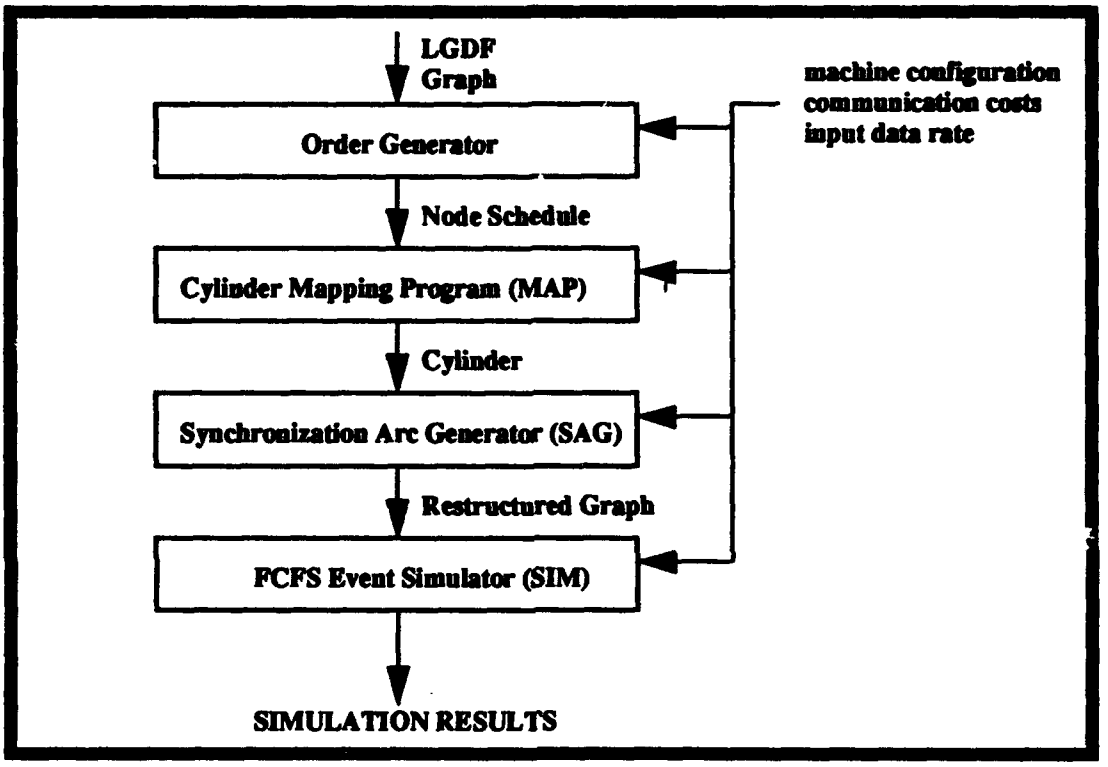

Figure 4.1. Program Usage to Produce Results

A. INITIAL TRIALS ON TEST GRAPH

The initial tests were performed on a simple data flow graph to generate baseline results for the Revolving Cylinder algorithm. This simple graph consisted of one input node and output node (execution time = 0), and 15 uniform instruction nodes (execution time = 10000). The nodes had no setup or breakdown latency, and an instruction size of zero. Therefore, the only communication is due to the transfer of data between processors and memory. The produce amount, consume amount, write amount, read amount, and threshold amount were all equal for a given queue. However, this number was different for the queues in the system (either 1000, 2000, or 4000 words). The queue capacity is eight times this amount. Several mappings

of this graph were used at various communication costs over three, four, and five processors. Figure 4.2 shows the test graph, with the numbers representing the quantities for the queues.

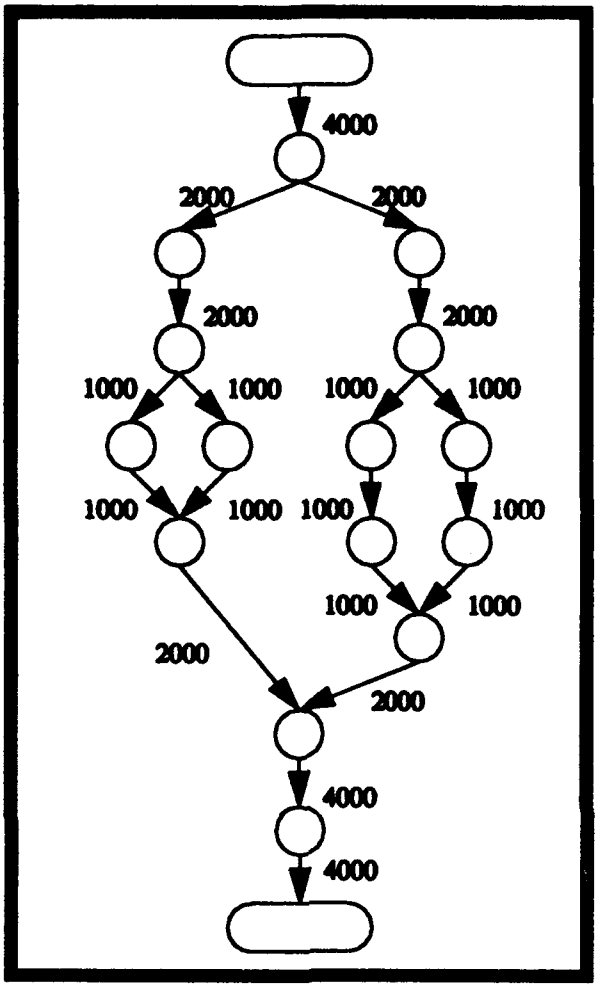

Figure 4.2. Test Data Flow Graph

The mappings of the nodes to processors for this graph was determined manually, attempting to maximize the communication / computation overlap. It is noted that in the all mappings for three processors, the processors were uniformly load balanced, each processor having exactly the same mapping (as far as computation and communication times) as the other two processors. The mappings for five processors were fairly well load balanced with exactly three nodes on each processor. However, the amount of communication overlap on each processor varied. The mappings on four processors were more difficult to determine as the nodes do not map evenly to processors.

All mappings were tested at four different communication costs, one, two, three, or four cycles to transfer one word of data between a processor and memory. The scheduler latency was set at zero. For this

graph, the yielded communication / computation ratios are 0.4, 0.8, 1.2, and 1.6 respectively. The simulation was set to compute the maximum throughput. Along with a First-Come-First-Serve (FCFS) test for the graph, each mapping was tested using four different variations of the Revolving Cylinder (RC) scheduling technique as described in the previous section: Start After Finish (SAF), Start After Start (SAS), Start After Finish with nodes bound to processors (SAFb), and Start After Start with nodes bound to processors (SASb).

In these tests, the number of memory modules was equal to the number of arithmetic processors in the system. All nodes mapped to a given processor were assigned to one memory module. The queues were assigned to the memory module to which their sink node is assigned. For FCFS tests, the same memory assignments were used as for the RC analysis to allow for direct comparison.

One important note must be made about the charts which follow. Although there are several mappings for each of the scheduling variations, only the result of the best mapping is shown. At different communication costs, the best mapping would often be different. Even at the same communication cost, various scheduling techniques could be better on different mappings.

The first test results were for a contention free situation. This is an ideal result where a node or queue is always able to access the memory unit where its required data is located. Figure 4.3 shows the results of the contention free test on three processesors.

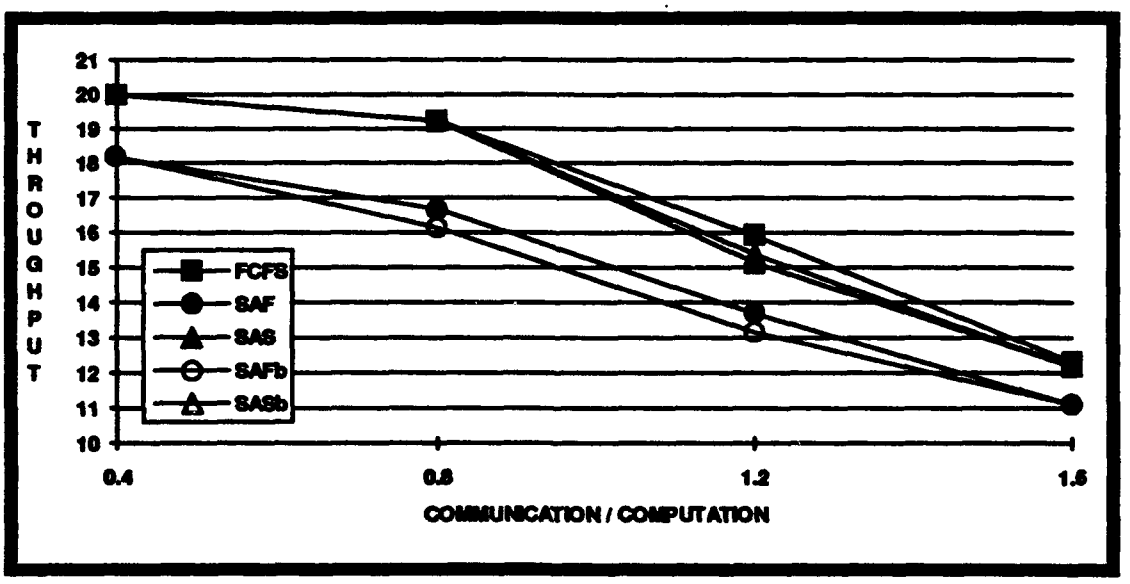

Figure 4.3. Test Graph on 3 Processors (Contention Free)

In this test, it is apparent that with no contention, FCFS provides the best throughput. SAS can come close to FCFS, but SAF is lacking, due to the inability to overlap consecutive cylinders. Processor binding yields no significant difference.

The next test is with memory contention a factor. Figures 4.4, 4.5, and 4.6 provide the results for three, four, and five processors respectively.

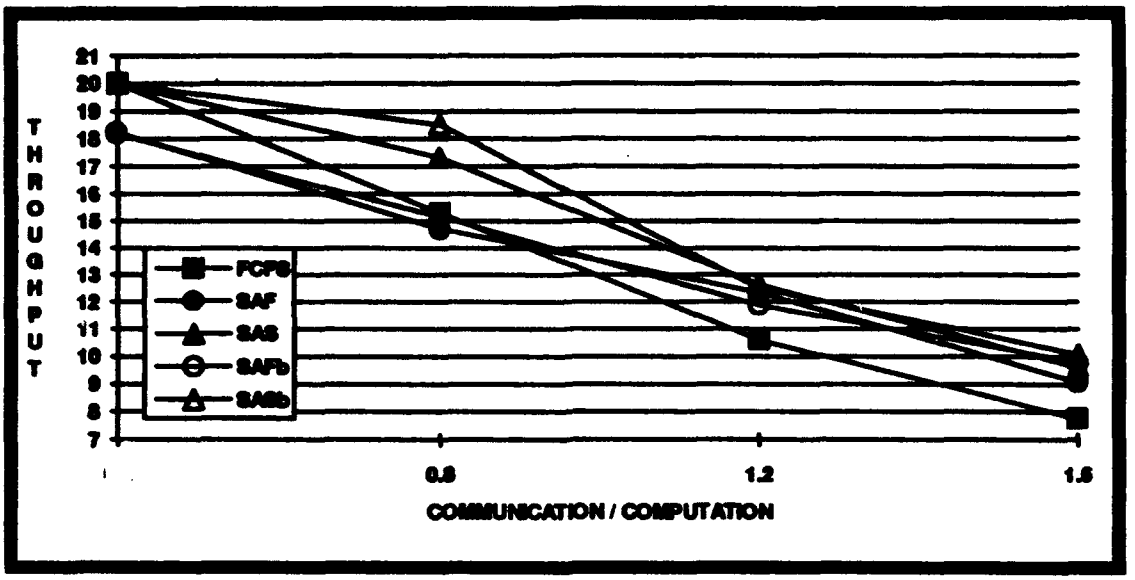

Figure 4.4. Test Graph on 3 Processors (with Contention)

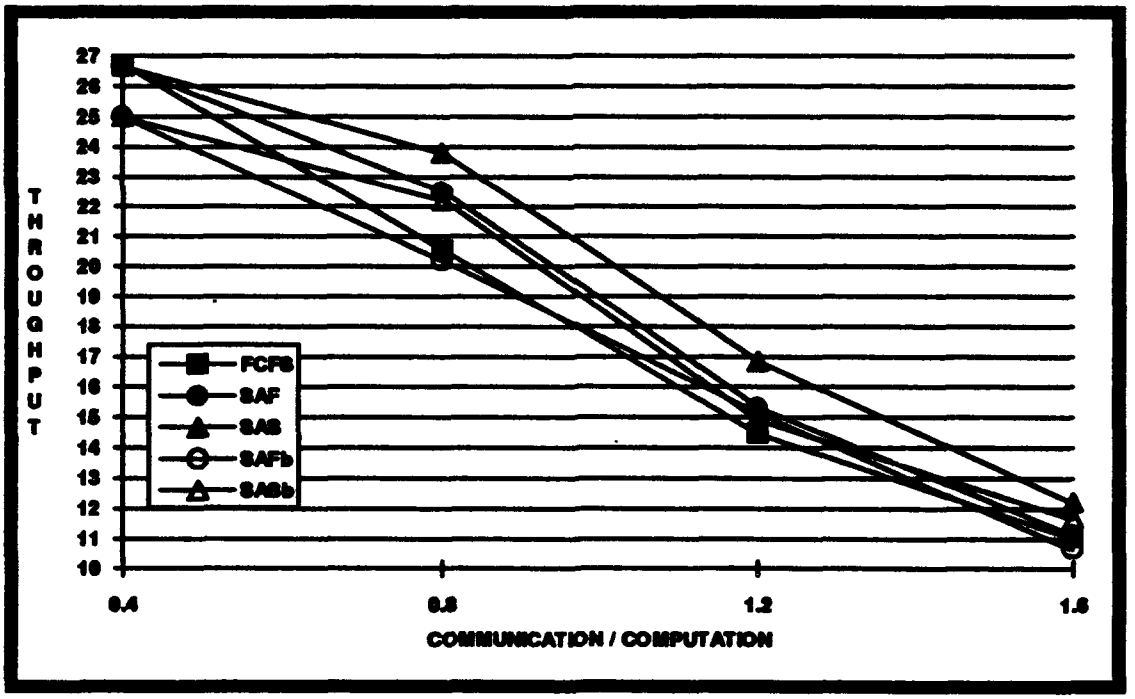

Figure 4.5. Test Graph on 4 Processors (with Contention)

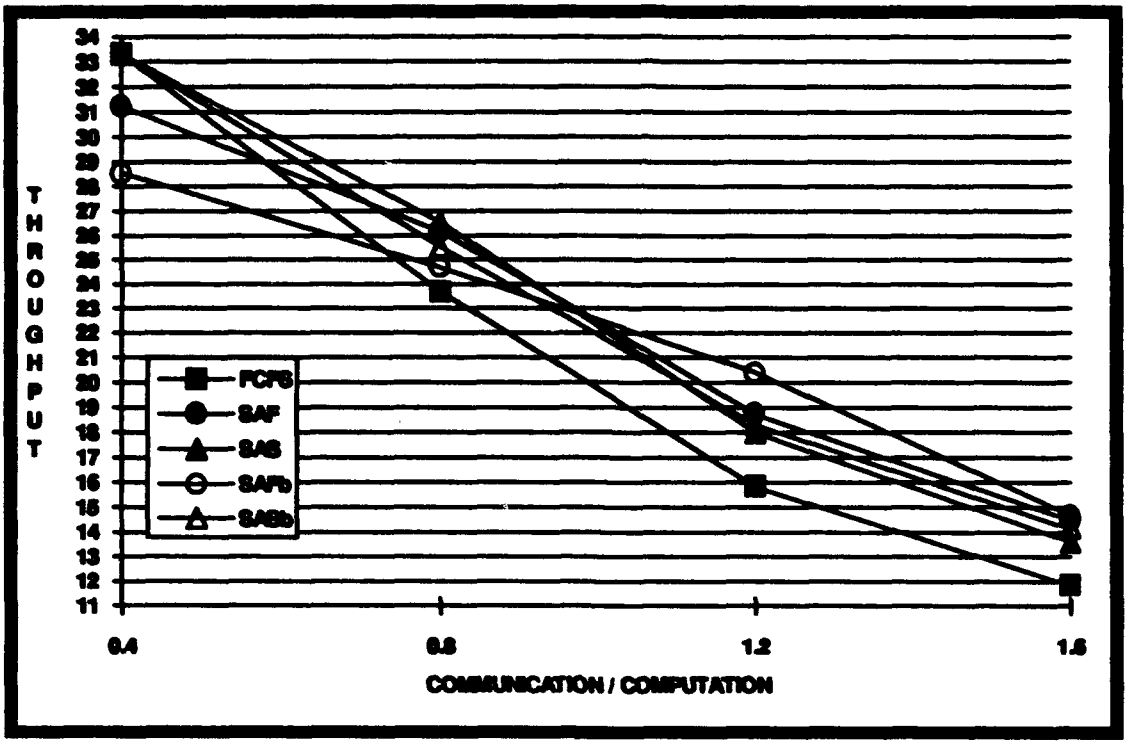

Figure 4.6. Test Graph on 5 Processors (with Contention)

Several points are apparent. At low communication costs, FCFS will provide high throughput. However, as the communication cost increases, FCFS throughput sharply decreases. The RC techniques show that increased throughput over FCFS is possible, especially as the communication cost increases. This is due to being able to map the nodes such that contention is minimized. However, as to determining the best variation of this technique, there is no consensus of results. Certain variations proved better for certain mappings. As stated previously, these charts show the best result for the scheduling variation. These results are not necessarily from the same mapping. Furthermore, only three or four mappings were used and there are many more mappings which are possible.

Figure 4.7 is a plot showing the effects of contention versus no contention for FCFS. It can be easily seen that contention is a major consideration, except at very low communication costs.

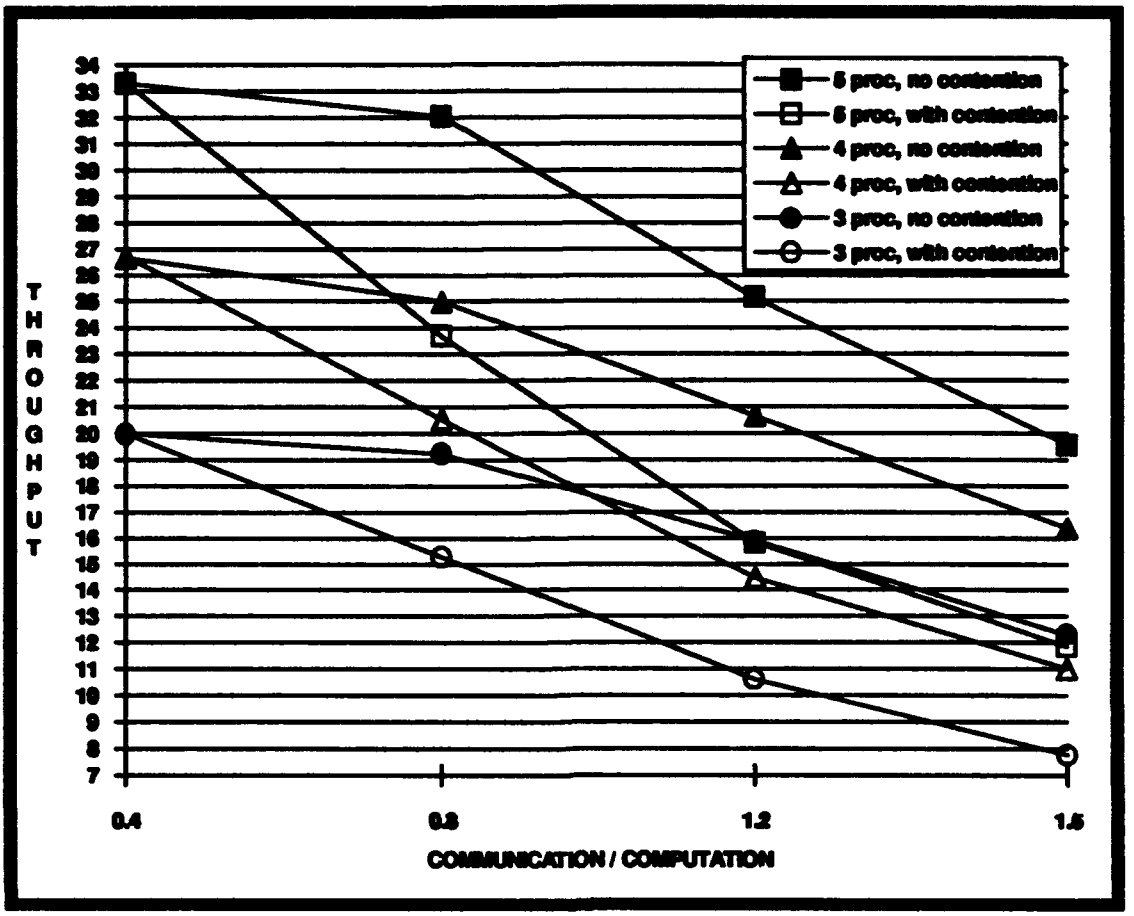

Figure 4.7. FCFS Contention versus No Contention

Figure 4.8 is a plot showing the effects of contention versus no contention for RC. Note that although contention still affects the throughput, the effects are much reduced compared with FCFS. As with the previous charts, the best results from RC are plotted.

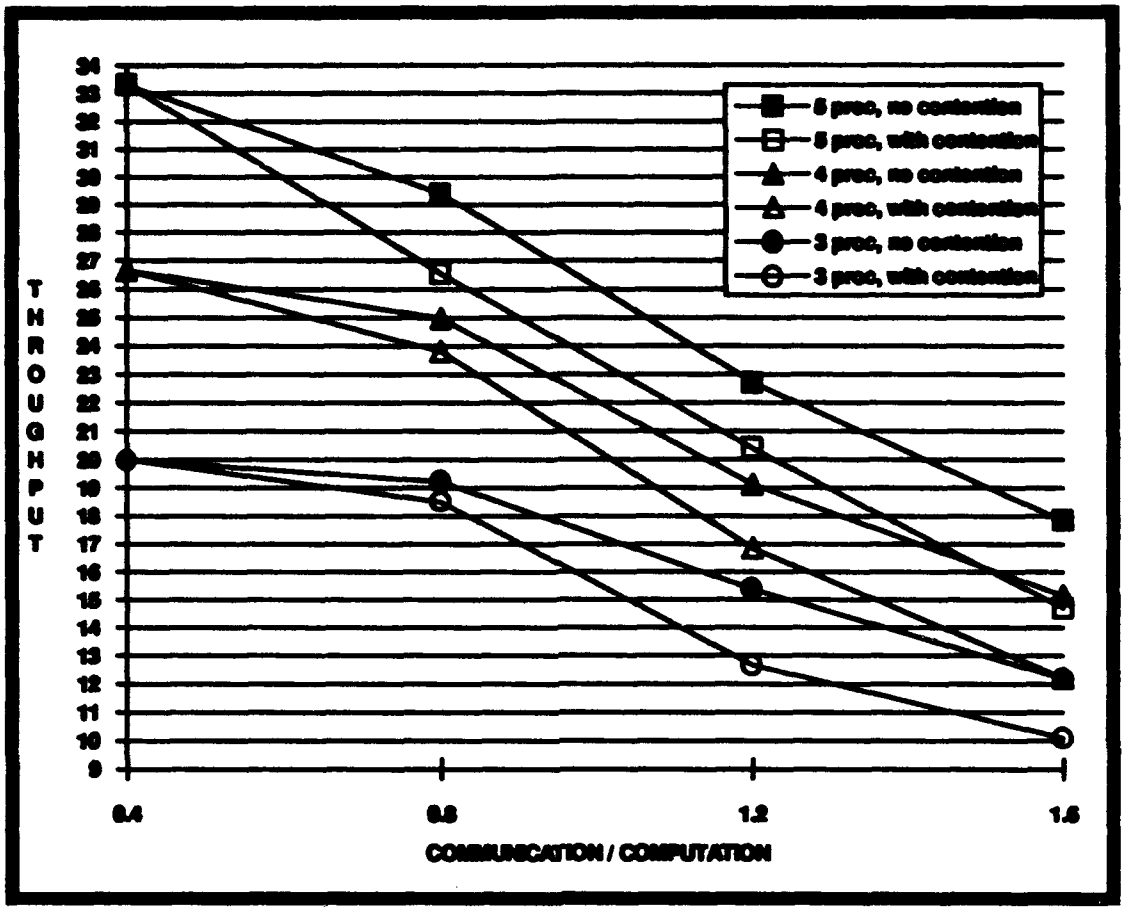

Figure 4.8. RC Contention versus No Contention

To demonstrate the improvements, Figure 4.9 is a plot comparing the contention free case and the contention case for both FCFS and RC. The 'Throughput Decrease' is the difference between the contention free and contention case divided by the throughput of the contention free case for the given number of processors and communication / computation ratio, and converted to a percentage. This percentage represents the degredation caused by adding memory contention to the model, with a higher figure representing higher degredation. As expected, it is seen that as the communication / computation ratio increases, the degredation

due to contention also increases. The number of processors plays only a small part in the ratio. An important note is that RC is not nearly as degraded by contention compared with FCFS.

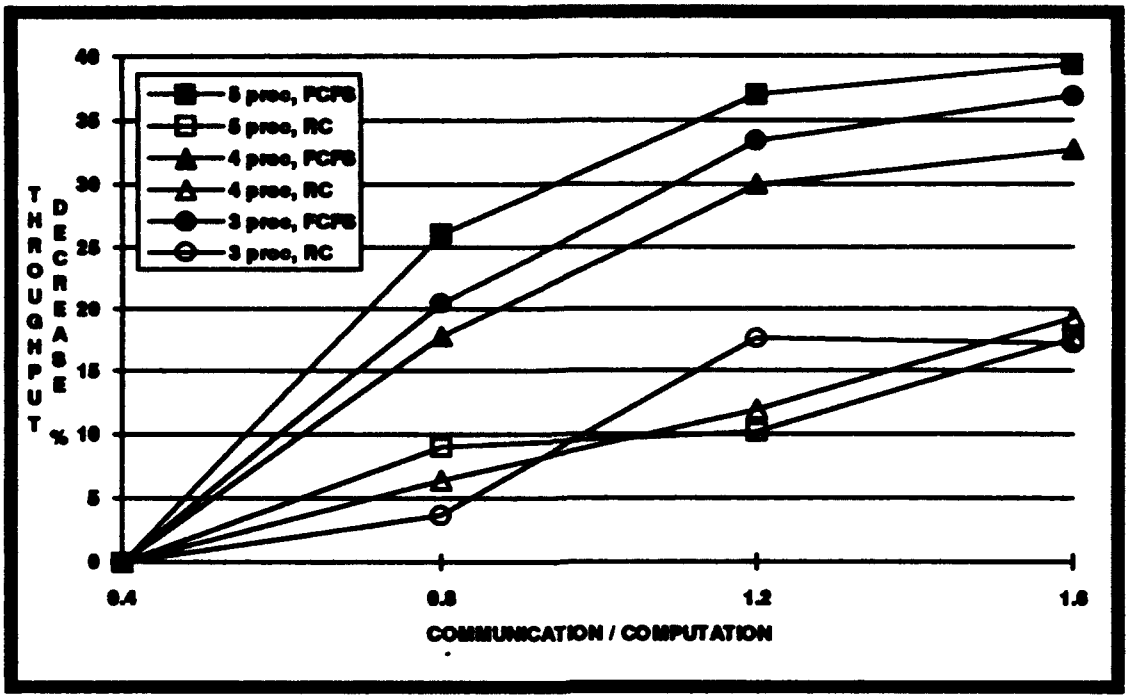

Figure 4.9. Throughput Decrease Due to Contention for FCFS and RC

B. TESTS ON AN ACTUAL APPLICATION GRAPH

The RC techniques were next practiced on an actual application graph. The graph chosen was the 'Active Sonobuoy' graph provided by AT&T for the ECOS simulator of the EMSP system [Ref. 13], and modified to fit the described system model. As with the test graph, the node setup and breakdown latencies are zero, the node instruction size is zero, and the scheduler latency is zero. The produce amount, consume amount, write amount, read amount, and threshold amount are the same for a given queue, with the capacity eight times this quantity. The number of memory modules is equal to the number of processors, with all nodes mapped to a processor assigned to the same memory and queues assigned to the memory of the sink node. The simulator is set to determine the maximum throughput of the system.

Figure 4.10 shows the active sonobuoy graph. The node execution times and queue quantities are given at the bottom for each 'level' of the graph, as all nodes and queues on each level are the same. The exception

is for the final 'level' of nodes where the execution time is below each node and the queue quantity for all queues into that node, as the quantities differ. Note this graph provides for a high degree of parallel execution.

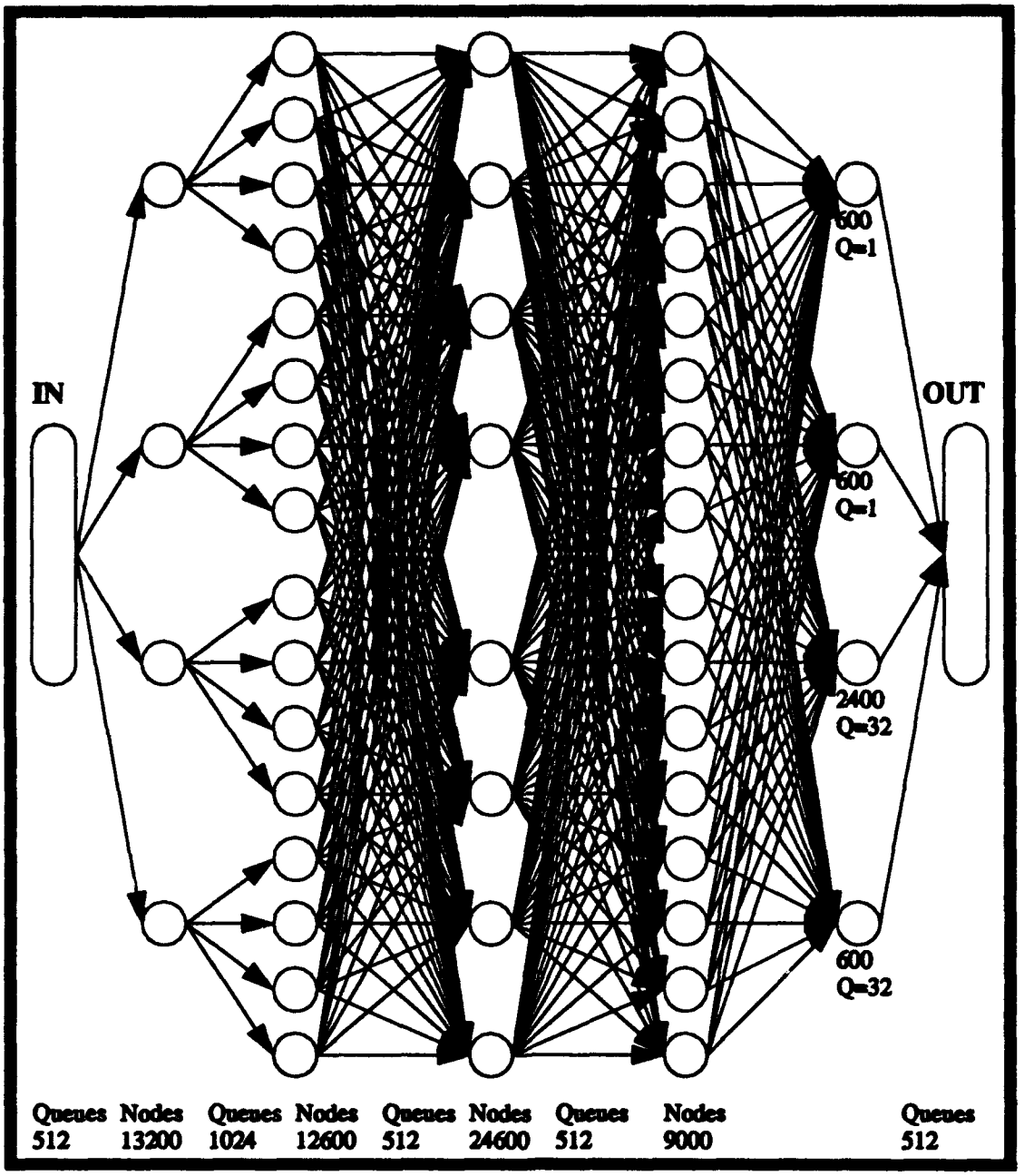

Figure 4.10. Active Sonobuoy Graph

Only one mapping on each of three different processor arrangements (four, eight, and thirteen) was tested. Yet in that small test sample, the results for this graph generally mirror the results for the test graph. With low communications costs, FCFS yields good results and there is no gain with RC. However, as

communication costs increase, RC can yield increasing improvements. Once again, there is no concurrence as to which variation of RC will consistently yield the best results. For exact results, see [Ref. 11].

C. ADDITIONAL RESULTS

In both of the graphs tested, another result viewed is the coefficient of variation. This is a measure of the regularity of completion, or response time of graph instances. The lower this number, the closer the response times of all the measured graph instances to the average response time. With both graphs, the coefficient of variation for RC is consistently less than FCFS. With some mappings, it is possible to reduce the coefficient of variation to zero. However, it must be noted that although RC is an improvement over FCFS, the results for FCFS were low to begin with.

V. CONCLUSION

This thesis provides a model for a Large Grain Data Flow (LGDF) computer system. This system utilizes two part processors, where one part handles communications and the other handles execution. The applications running on this computer system are modeled as data flow graphs consisting of nodes and queues.

A scheduling technique known as the Revolving Cylinder (RC) is described, with four variations. In tests versus simple First-Come-First-Serve (FCFS) scheduling, it is shown that RC can lead to increased throughput, especially as communication costs increase. However, it is seen that selecting the appropriate mapping is not a simple task, and a good mapping for one communication cost is not necessarily a good mapping for another communication cost. It is also shown that none of the variations of RC are consistently better than any other variation, and are dependent on the mapping.

A. EXPANDED TESTING

In this research, the purpose was to generate baseline results which allow for further expansion. Many additional tests must be conducted to fully analyze the effectiveness of the RC technique. Several important issues must be studied.

For nodes, in all tests, the instruction size is zero. Therefore, there is no memory contention associated with retrieval of the instructions from memory. The input and output nodes have no bearing on processing with the execution time set to zero.

For queues, in all cases the produce/consume, read/write, and threshold amounts were always constant. Varying these quantities could have a major impact on graph execution.

All latencies, node setup and breakdown, and scheduler latency were zero. This reduces the communication overhead.

All tests were made with the number of memory modules equal to the number of arithmetic processors. Tests need to be made with varying numbers of memory modules to fully analyze the effects of memory contention.

All tests were based on maximum graph throughput. Tests need to be completed with various graph activation rates.

B. FUTURE RESEARCH

The primary area for future research work regarding RC is in the area of mapping. The results of this paper show that a mapping for RC can be found which improves performance over FCFS. However, there is no method for easily obtaining this mapping due to the many variables involved. Accurate characterization of the cylinder mapping is necessary to develop a metric for a good mapping. This would imply establishing a correlation between a given mapping and its run-time performance.

LIST OF REFERENCES

1. Brobst, S. A., "Organization of an Instruction Scheduling and Token Storage Unit in a Tagged Token Data Flow Machine," in *Proceedings of the 1987 International Conference on Parallel Processing, vol. 3*, August 1987.

2. Lee, E. A., and Messerschmitt, D. G., "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," in *IEEE Transactions on Computing, vol. C-36, no. 1*, January 1987.

3. Karp, R. M., and Miller, R. E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," in *Journal of Applied Mathematics, vol. 14, no. 6*, November 1966.

4. Lee, E. A., "Consistency in Dataflow Graphs," in *IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 2*, April 1991.

5. Gurd, J. R., Kirkhame, C. C., and Watson, I., "The Manchester Prototype Dataflow Computer," in *Communications of the ACM*, January 1985.

6. Lee, E. A., and Bier, J. C., "Architectures for Statically Scheduled Dataflow," in *Journal of Parallel and Distributed Computing, vol. 10*, December 1990.

7. King, C. T., Chou, W. H., and Ni, L. M., "Pipelined Data - Parallel Algorithms: Part I - Concept and Modeling," in *IEEE Transactions on Parallel and Distributed Systems*, October 1990.

8. Shukla, S. B., Little, B. S., and Zaky, A., "A Compile-time Technique for Controlling Real-time Execution of Task-level Data-flow Graphs," presented at the 1992 International Conference on Parallel Processing.

9. Rau, B. R., Glaeser, C. D., and Picard, R. L., "Efficient Code Generation for Horizontal Architectures: Compiler Technique and Architectural Support," in *Proceedings of the 9th International Symposium on Computer Architecture*, 1982.

10. Rice, M. L., "The Navy's New Standard Digital Signal Processor: The AN/UYS-2," paper presented at the Association of Scientists and Engineers 27th Annual Technical Symposium, 23 May 1990.

11. Naval Postgraduate School Technical Report NPS-EC-93-015, *Revolving Cylinder Analysis: A Technique for Restructuring of Large Grain Data Flow Graphs Representing Throughput-Critical Applications*, by D. M. Cross, S. B. Shukla, and A. Zaky, September 1993.

12. Naval Postgraduate School Technical Report NPS-EC-93-016, *A Tool for the Analysis of the Parallel Execution of Throughput-Critical LGDF Programs: A User Manual*, by D. M. Cross, S. B. Shukla, and A. Zaky, September 1993.

13. AT&T Technologies, Report IN 48280, *ECOS Workstation Tutorial*, AT&T Bell Laboratories, 30 March 1991.

14. Akin, C., *A Periodic Input Processing Data Flow Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.

15. Bell, H. A., *A Compile-Time Approach for Chaining and Execution Control in the AN/UYS-2 Parallel Signal Processor*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1992.

16. Little, B. S., *A Technique for Predictable Real-Time Execution in the AN/UYS-2 Parallel Signal Processing Architecture*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1991.

17. Swank, D., *Large Grain Data-Flow Graph Restructuring for EMSP Signal Processing Benchmarks on the ECOS Workstation System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1993.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Chairman, Code CS Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5118	1
5.	Prof. Shridhar B. Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	2
6.	Prof. Amr Zaky, Code CS/Za Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5118	2
7.	Mr. David Kaplan Naval Research Laboratory 4555 Overlook Avenue, SW Washington, DC 20375-5000	1
8.	Mr. Richard Stevens Naval Research Laboratory 4555 Overlook Avenue, SW Washington, DC 20375-5000	1

9. Commander, Naval Sea Systems Command — 1
PMS 428
Naval Sea Systems Command Headquarters
Washington, DC 20362-5101

10. American Telephone and Telegraph Bell Laboratories — 1
Attn: Mr. Jerome Uhrig, WH 46243
67 Whippany Road
P.O. Box 903
Whippany, NJ 07981-0903

11. CPT David M. Cross, USA — 1
444 Arbor Road
Cinnaminson, NJ 08077